COMPUTERBOOKS

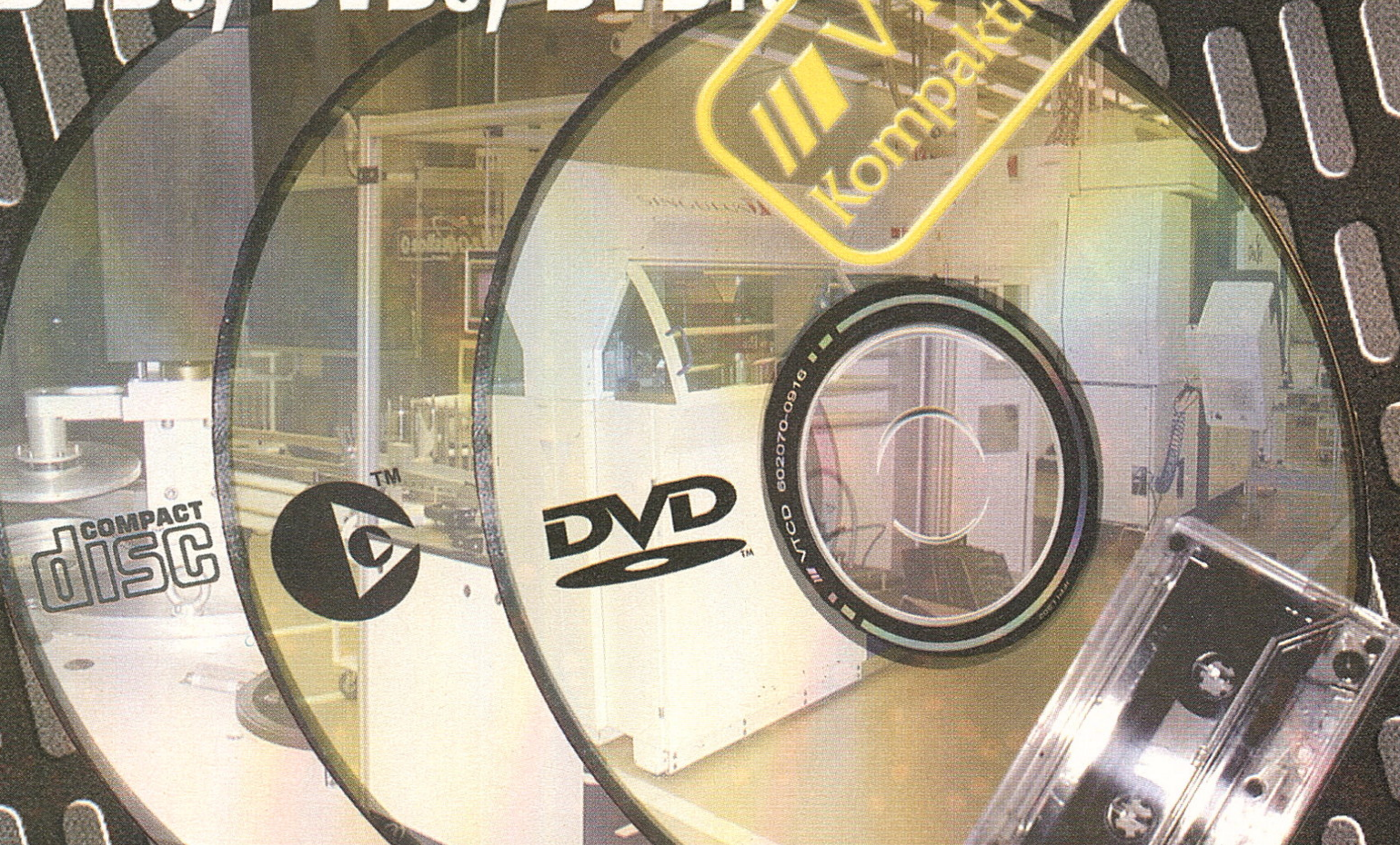
Együtt könnyebb a programozás

Objektum-orientált C++

MINDENT EGY HELYEN

authoring, ÚJ
mastering, sokszorosítás...
DVD5, DVD9, DVD10

VTCD VIDEOTON
Kompaktlemez-gyártó Kft.



...továbbra is az Ön partnere vagyunk CD, MC gyártásban.

H-8000 Székesfehérvár, Pf.:175.
Tel.:(22) 533-571, Fax:(22) 533-077
vtcd@vtcd.hu www.vtcd.hu

DVD Authoring studio:
H - 1021 Budapest, Hűvösvölgyi út 54,
Tel. : (1) 392-1217, Fax : (1) 392-1238
E-mail: authoring@vtcd.hu www.vtcd.hu

TÜV
Rheinland EUROQUA
EN ISO 9001:2000
ZN 75.100.8943

Benkő Tiborné
Poppe András

Együtt könnyebb a programozás

Objektum-orientált C++

LEKTOR
TÓTH BERTALAN

COMPUTERBOOKS
BUDAPEST, 2006

A könyv készítése során a Kiadó és a Szerzők a legnagyobb gondossággal jártak el. Ennek ellenére hibák előfordulása nem kizárható.
Az ismeretanyag felhasználásának következményeiért sem a Szerzők, sem a Kiadó felelősséget nem vállal.

Minden jog fenntartva. Jelen könyvet vagy annak részleteit a Kiadó engedélye nélkül bármilyen formátumban vagy eszközzel reprodukálni, tárolni és közölni tilos.

©Benkő Tiborné, Dr. Poppe András
Társszerző: Tóth Bertalan , javított kiadás 2005. április

Harmadik kiadás, 2006 február

ISBN: 963 618 324 4

©Kiadó:
ComputerBooks Kiadó Kft.
1126 Bp., Tartsay Vilmos u. 12.
Telefon: 375-1564; Tel/Fax: 375-3591
E-mail: info@computerbooks.hu
http://www.computerbooks.hu

Felelős kiadó: ComputerBooks Kft. Ügyvezetője
Borítóterv: Székely Edith

Nyomdai munkák: Pressman Bt.

Tartalomjegyzék

Bevezetés 1

I. A C++ MINT EGY JOBB C NYELV 3

1. C++ program szerkezete 5
- 1.1 Az első C++ program 5
- 1.2 Megjegyzések használata 6
- 1.3 Több modulból álló C++ program 6

2. Alaptípusok, változók és konstansok 7
- 2.1 A C++ nyelv típusai 7
  - 2.1.1 Típuselőírások, típusmódosítók 8
  - 2.1.2 A felsorolt típus (enum) 8
  - 2.1.3 Saját típusok előállítása 9
  - 2.1.4 Konstansok a C++ nyelvben 10
- 2.2 Értékek, címek, mutatók 10
- 2.3 A void* típusú általános mutató 11
- 2.4 Többszörös indirektségű mutatók 12

3. Operátorok és kifejezések 13
- 3.1 Aritmetikai operátorok 13
- 3.2 Léptető operátorok 14
- 3.3 Bitműveletek 15
  - 3.3.1 Biteltoló műveletek 16
- 3.4 A sizeof operátor 17
- 3.5 A vessző operátor 18
- 3.6 A fetételes operátor 18
- 3.7 Az érvényességi kör (hatókör) operátor 19
- 3.8 Statikus és dinamikus helyfoglalású változók használata 20
  - 3.8.1 A new és a delete operátorok használata 21
- 3.9 Konstansok használata 22
- 3.10 Saját típus előállítása 23
- 3.11 Szabványos I/O használata 23

4. A C++ nyelv utasításai 29
- 4.1 Utasítások és blokkok 29
- 4.2 Az if utasítás 29
- 4.3 A switch utasítás 34
- 4.4 Ciklusutasítások 38
  - 4.4.1 A while ciklus 38

4.4.2 A for ciklus 40
4.4.3 A do-while ciklus 43

5. Származtatott típusok 45
5.1 Tömbök, sztringek és mutatók 45
5.1.1 Egydimenziós tömbök 45
5.1.2 Többdimenziós tömbök 49
5.1.3 Karaktersorozatok (sztringek) 50
5.2 Felhasználó általdefiniált adattípusok 51
5.2.1 A struct struktúratípus 51
5.2.2 Struktúratömbök 53

6. Függvények 59
6.1 Függvények definíciója és deklarációja 59
6.2 A függvények paraméterezése és a függvényérték 59
6.3 A függvényhívás 61
6.4 Különböző típusú paraméterek 64
6.4.1 Aritmetikai típusú paraméterek 64
6.4.2 Felhasználói típusú paraméterek 65
6.4.3 Tömbök átadása függvénynek 66
6.4.4 Sztringparaméterek 72
6.4.5 A függvény, mint argumentum 73
6.4.6 Alapértelmezés szerinti (default) argumentumok 74
6.4.7 Változó hosszúságú paraméterlista 75
6.4.8 A main() függvény paraméterei és visszatérési értéke 76
6.4.9 Rekurzív függvények használata 77
6.4.10 Inline függvények 78
6.4.11 Függvénynevek átdefiniálása (overloading) 79
6.4.12 Általánosított függvények (template) 80

7. Névterek és modulok használata 83
7.1 Saját névterek kialakítása és használata 83
7.2 Az extern tárolási osztály 85
7.3 Több modulból álló C++ program 87

8. Algoritmusok programozása 91
8.1 Osztók, legnagyobb közös osztó, legkisebb közös többszörös keresése 91
8.2 Rendezési algoritmusok 97
8.3 Interpolációk 99

9. Láncolt lista 103
9.1 Előre-hátra láncolt lista 103

10. Fájlkezelés struktúrák felhasználásával 107
10.1 Struktúrák írása állományba és olvasása állományból 107

II. AZ OBJEKTUM-ORIENTÁLT C++ NYELV 111

11. Bevezetés az objektum-orientált C++ nyelvbe 113
11.1 Mintafeladat: Műveletvégzés struct használatával 113

12. Osztályok 117
12.1 Az osztály tagjainak elérése 118
12.2 Az osztályok friend mechanizmusa 122
12.3 Statikus helyfoglalású objektumpéldány 123

13. Konstruktorok és destruktorok 125
13.1 Konstruktorok 125
13.2 Destruktorok 128

14. Operátorok túlterhelése (operator overloading) 133
14.1 Az osztályok bővítése input/output műveletekkel 133

15. Az öröklés (öröklődés) mechanizmusa 137
15.1 Származtatott osztályok 137
15.2 Korai kötés - bezáródás jelensége 140
15.3 Késői kötés - virtuális tagfüggvények 141
15.4 Öröklés több osztálytól 143
15.5 Öröklés helyett beágyazás 146

16. Általánosított osztályok (templates) 149
16.1 Specializáció és példányosítás 149
16.2 Sablonok alkalmazása öröklött objektumok esetén 151

17. Objektum-orientált láncolt-lista 153
17.1 Lista készítése előre-hátra láncolással 153
17.2 Listaelem törlése a láncból 157
17.3 Listaelem beszúrása a láncba névsor szerint 162

18. Numerikus módszerek objektum-orientált felfogásban 165
18.1 Interpoláció 165
18.2 Integrálás 168
18.3 Gyök meghatározása érintő módszerrel 171

19. Egy osztállyal feljebb léphetünk? 173
19.1 A C++ nyelv bővítése a halmaztípussal 173
19.2 Saját sztring típus és a hozzá tartozó műveletek definiálása 187
19.3 A szabványos sablontár (STL) alkalmazása 198
19.4 A fastruktúra (tree) megvalósítása 204
19.5 Kettős lista használata 211

FÜGGELÉK 219

F1. A Dev-C++ fejlesztői környezet használata 221
F1.1 Objektum-orientált program állományai 221
F1.2 Új project létrehozása 221
F1.3 A program futtatása 224
F1.4 Project betöltése 224

F2. C++ óravázlatok 225
1. Gyakorlat 227
2. Gyakorlat 234
3. Gyakorlat 240
4. Gyakorlat 245
5. Gyakorlat 251
6. Gyakorlat 257
7. Gyakorlat 267
8. Gyakorlat 275
9. Gyakorlat 284
10. Gyakorlat 293
11. Gyakorlat 299
12. Gyakorlat 307
13. Gyakorlat 320
14. Gyakorlat 330
15. Gyakorlat 338
16. Gyakorlat 348

F3. A CD-melléklet tartalomjegyzéke 351

Tárgymutató 353

Bevezetés

"*Még egy programozási könyv!*" - gondolhatja a könyváruház informatikai részlegében barangoló Vásárló. Ha azonban már a kezében tartja ezt a könyvet, megérti, hogy kicsit elsietett volt az észrevétele. Amit a kezében tart valójában egy C++ tankönyv, ami egyaránt segíthet a tanulóknak, hogy akár önállóan elsajátítsák a C++ nyelv alapjait, és segíthet az oktatóknak is, hisz a tananyag az alapkönyvet kiegészítő, 16 gyakorlati foglalkozásra bontva, óravázlatként is felhasználható.

Könyvünk alapvető célja, hogy feladatokon keresztül megismertesse az objektum-orientált C++ nyelv használatát. A feladatokhoz rövid elméleti magyarázatot fűztünk, azonban részletesebb leírást talál az Olvasó a ***ComputerBooks Kiadó*** egy másik kiadványában, a ***Programozzunk C++ nyelven!*** című könyvben.

Az egyes fejezetekben a programkészítés módszerének elsajátításához mutatunk be gyakorló feladatokat, melynek megértése és a kiírt feladatok megoldása hozzájárul az önálló tanuláshoz. Csak röviden világítunk rá azokra az elméleti ismeretekre, amelyek feltétlenül szükségesek a feladatok megoldásához.

A C++ nyelv támogatja a hagyományos, algoritmikus programkészítést, ezért könyvünk első „**A C++ mint egy jobb C nyelv**" része is ezt a megközelítést tűzte a zászlajára. A C nyelvet ismerő Olvasók nagy lépésekkel haladva dolgozhatják fel az ide tartozó fejezeteket, a kezdő programozók pedig elsajátíthatják a C++ nyelv alapjait (típusok, definíciók, utasítások, függvények).

A könyv második „**Az objektum-orientált C++ nyelv**" része a napjainkban nagyon elterjedt programfejlesztési módszerrel, az objektum-orientált programozással ismerteti meg az olvasót. Lépésről-lépésre haladva feltárulnak az osztályok, azok különböző tagjai, az öröklés és végül a nagyobb rendszerek fejlesztéséhez nélkülözhetetlen sokalakúság.

Fontos része a műnek a **függelék**, és ez a megállapítása nemcsak az oldalszámra, hanem a tartalomra is vonatkozik. Az F2. függelék olyan oktatási anyagot tartalmaz, amely gyakorlati foglalkozásonként csoportosítja a különböző témaköröket. A feladatlapok PDF-formátumban a CD-mellékleten is megtalálhatók. A feladatsorok felölelik a könyv teljes tartalmát, azonban más feladatokon keresztül mutatják be a C++ lehetőségeit, rendszerezetten végighaladva az alapismereteken. Minden gyakorlati foglalkozás elején röviden ismertetjük az aktuális témakört és a hozzá tartozó mintafeladatokat. A kidolgozott mintafeladatok nagyban segíthetik a programozási lépések, fogások hatékony elsajátítását.

A kiírt gyakorlati feladatok három részre oszthatók:

- Némely feladathoz ajánlott deklarációt és definíciót javasolunk, illetve megadjuk az elkészítendő függvények prototípusát. Egyes feladatoknak várható futási eredményét is közöljük.
- Vannak olyan feladatok is, amelyek részmegoldást tartalmaznak, azonban a befejezéshez bizonyos programrészeket önállóan kell elkészíteni.
- Néhány feladatnak csak a kiírása szerepel, ezeket teljesen önállóan kell megoldani.

A CD-melléklet a könyv anyagához tartozó programok forráskódján és bizonyos elektronikus formátumú fejezeteken túlmenően egy szabadon terjeszthető C/C++ fejlesztőeszközt is tartalmaz. A ***\DevCpp\devcpp4970-gcc295.exe*** állományával Windows alatt futtatható, ***ANSI C/C++*** fejlesztőeszköz telepíthető a rendszerünkbe.

Minden tanulási folyamat fontos része az önellenőrzés, amihez a CD-melléklet ***\Teszt*** könyvtárában elhelyezett, a *Windows98/2000/XP* alatt működő tesztprogram, illetve az itt található feladatsorok nyújtanak hathatós segítséget.

Köszönetnyilvánítás

Ezúton szeretnénk köszönetet mondani azoknak a villamosmérnök és gépészmérnök hallgatóknak, akik végighallgatták a függelékben kidolgozott oktatási anyag módosított változatát. Az észrevételeiket és a javaslataikat azzal szeretnénk még megköszönni, hogy Kar szerinti bontásban, névsor szerint közzétesszük a neveiket:

Villamosmérnök hallgatók	*Gépészmérnök hallgatók*
Hegedüs Attila	***Jusztin Krisztián Gábor***
Kenderes Tamás	***Krisch János***
Kovács Olivér	***Kunkli Péter***
Molnár Gábor József	
Perjési Máté	
Ürmös Antal	

Továbbá, köszönetet mondanunk ***Szilárd Lászlónak***, *BME Gépészeti Informatika Tanszék* munkatársának, az öröklés fejezet kidolgozását segítő értékes javaslataiért, valamint ***Lehel György*** programozónak az oktatási anyaghoz fűzött megjegyzéseiért.

A Szerzők

I. A C++ mint egy jobb C nyelv

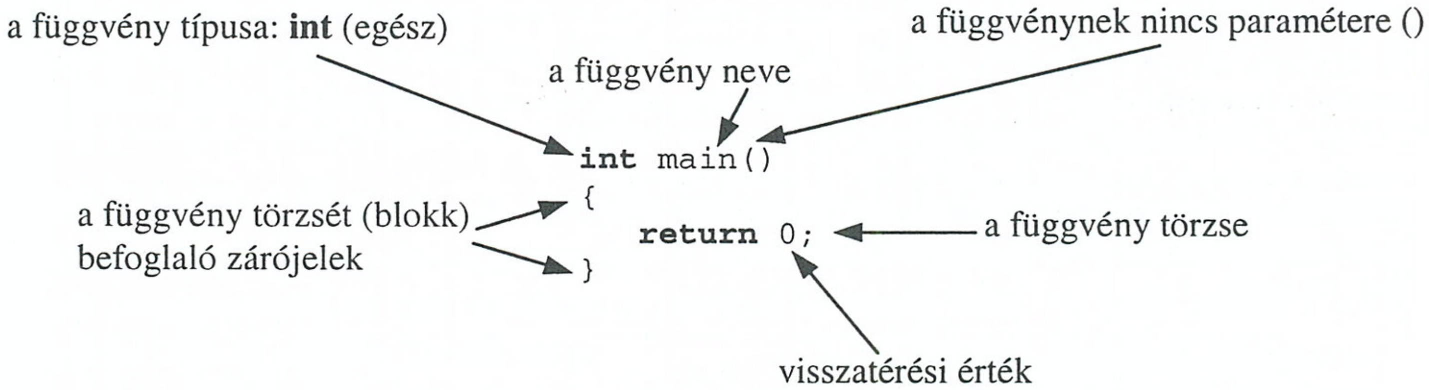
1. C++ program szerkezete

Ebben a fejezetben egyszerű példákon keresztül bemutatjuk a C++ program szerkezetét.

1.1 Az első C++ program

Minden C++ program egy vagy több függvényből épül fel. A függvények közül egyet kötelezően a ***main*** névvel kell ellátni.

A legegyszerűbb C++ program egyetlen üres ***main()*** függvény:



```
int main()
{
    return 0;
}
```

a függvény típusa: int (egész)
a függvény neve
a függvénynek nincs paramétere ()
a függvény törzsét (blokk) befoglaló zárójelek
a függvény törzse
visszatérési érték

> Írjunk programot, amely szöveget ír ki a képernyőre, és a szöveg után sort emel! A program várjon egy karakter leütésére! (*Elso*)

A C++ nyelvben a szabványos I/O műveletek végzésére használt ***cin*** és ***cout*** adatfolyam (*stream*) objektumok nem részei a C++ nyelv definíciójának. A C++ szabvány minden könyvtári elemet egy közös ***std*** névterületen definiál. A szabványos outputként a ***cout***, a szabványos input elvégzésére pedig a ***cin*** adatfolyam-objektumot használjuk. A szöveg kiírása "" között történik, és új sorba megy a kurzor az ***endl*** hatására.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Elso C++ program!" << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elso C++ program!
```

1.2 Megjegyzések használata

Írjunk programot, amely két sorban ír ki szöveget a képernyőre, és az utasításokat kétféle megjegyzéssel magyarázza! (*Masodik*)

A kiíráskor a szövegben elhelyezett `'\n'` karakter is sort emel. Ha a szöveg elején adjuk meg: `"\n szoveg"`, akkor a *szoveg* kiírása előtt, ha pedig a végén: `"szoveg\n"`, akkor a *szoveg* kiírása után fog sort emelni. Természetesen lehet a szövegben bárhol, akkor tördeli a szöveget.

A /* */ között többsoros megjegyzés írható, a // használatával pedig a sor végéig terjedő megjegyzés adhatunk meg.

```
#include <iostream>
using namespace std; // közös névterület
/* Ez a program bemutatja a
megjegyzések használatát. Ha a megjegyzés
hosszabb egy sornál, akkor érdemes a
bemutatott megoldást használni. */
int main()
{
  //szöveg kiírása két részben
  cout << "Masodik" << " C++ program!\n"; // új sor '\n' használata
  cout << "Megjegyzesek \nhasznalata\ntobb\nsorban." << endl;
  cin.get(); // karakter leütésére vár
  return 0; // visszatérés 0 értékkel
}
```

A program futásának eredménye:

```
Masodik C++ program!
Megjegyzesek
hasznalata
tobb
sorban.
```

1.3 Több modulból álló C++ program

A C++ nyelv támogatja a moduláris programozás elvét. A moduláris programozás előnye, hogy a modulok külön is lefordíthatók, így csak azok a modulok fordulnak újra, amelyekben változás történt. Ez a módszer jelentősen csökkenti a program fordításának idejét. A modulokkal érvényesül az adatrejtése elve.

Több modulból álló alkalmazást találunk a könyvünk 7.3. fejezetében.

A példatár programjai a CD-mellékleten található ***Dev-C++4.9.7.0*** szabadon terjeszthető fordítóval készültek.

2. Alaptípusok, változók és konstansok

A C++ nyelvben a program számára mindig deklarálnunk kell a használni kívánt neveket. A deklaráció során csak a név tulajdonságait adjuk meg a fordítónak. *Definíciót* kell használnunk, ha azt szeretnénk, hogy az adott deklarációnak megfelelő memóriafoglalás is végbemenjen, tehát a *definíció* olyan *deklaráció*, amely helyfoglalással jár. A név definíciója csak egyszer szerepelhet a programban, míg ugyanazt a nevet többször is deklarálhatjuk, de az egymást követő deklarációknak egyezniük kell.

2.1 A C++ nyelv típusai

Definiáljunk **int**, **float** és **double** típusú változókat, adjunk kezdőértéket, és írassuk ki a tartalmukat szöveges formában. (*Pelda_01*)

Ha meg akarjuk jeleníteni egy változó tartalmát, akkor ***cout*** adatfolyamban a << operátor után a változó nevét kell megadni. A szöveget idézőjelek ("") között adjuk meg, amely magyarázatul szolgál az utána megjelenő eredménynek. Az ***endl*** hatására a következő kiírás új sorban jelenik meg.

```
#include <iostream>
using namespace std;

int main()
{ int i = 120000, j = -13;
  float a = 3.56, b = -2.586666;
  double x1 = 1.6e7,x2 = 2.4e5, y = 5e-2;
  cout << " i (120000)    = " << i << endl;
  cout << " j (-13)       = " << j << endl;
  cout << " a (3.56)      = " << a << endl;
  cout << " b (-2.586666) = " << b << endl;
  cout << " x1(1.6e7)     = " << x1 << endl;
  cout << " x2(2.4e5)     = " << x2 << endl;
  cout << " y (5e-2)      = " << y << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
 i (120000)    = 120000
 j (-13)       = -13
 a (3.56)      = 3.56
 b (-2.586666) = -2.58667
 x1(1.6e7)     = 1.6e+07
 x2(2.4e5)     = 240000
 y (5e-2)      = 0.05
```

2.1.1 Típuselőírások, típusmódosítók

Definiáljunk különböző típusú változókat, használjuk a **short** és a **long** tárolási hossz-, valamint az **unsigned** előjel-módosítót! A definícióban adjunk kezdőértéket, majd írassuk ki a változók tartalmát szöveges formában! (*Pelda_02*)

```
#include <iostream>
using namespace std;
int main()
{ short int i = 32000;
  unsigned short j = 65000;
  char c1 = 65;
  unsigned char c2 = 82;
  unsigned long h = 123456789;
  cout << " i  (32000)     = " << i << endl;
  cout << " j  (65000)     = " << j << endl;
  cout << " c1 (65)        = " << c1 << endl;
  cout << " c2 (82)        = " << c2 << endl;
  cout << " h  (123456789) = " << h << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
 i  (32000)     = 32000
 j  (65000)     = 65000
 c1 (65)        = A
 c2 (82)        = R
 h  (123456789) = 123456789
```

2.1.2 A felsorolt típus (enum)

Az **enum** típust a konstansnevek felhasználásával hozzuk létre:

enum *azonosító* {*felsorolás*};

Deklaráljunk **enum** típusú konstansokat, állítsunk be a mezőknek különböző értéket, amely megfelel a sorszámnak! Definiáljunk változókat, amelyek bemutatják az **enum** konstansok használatát! Írassuk ki a változók értékét szöveges formában! (*Enum*)

```
#include <iostream>
using namespace std;
enum tantargy{magyar, angol, olasz, kinai, japan};
enum uticel{Praga, Berlin = 3, Budapest=12};
int main()
{
  enum tantargy t1, t2;    // C, C++
  uticel hoval, hova2;     // C++
  t1 = olasz; t2 = magyar;
  cout << "Tantargy - olasz : " << t1 << endl;
  cout << "Tantargy - magyar: " << t2 << endl;
  hova1 = Praga;
  hova2 = Berlin;
  cout << "Uticel - Praga   : " << hova1 << endl;
  cout << "Uticel - Berlin  : " << hova2 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Tantargy - olasz : 2
Tantargy - magyar: 0
Uticel - Praga   : 0
Uticel - Berlin  : 3
```

2.1.3 Saját típusok előállítása

A **typedef** speciális tárolási osztály, amely lehetővé teszi, hogy létező típusokhoz szinonim neveket rendeljünk, és ezt követően már a szinonim névvel is definiálhatunk változókat.

typedef *létező_típus saját_típus*;

A **typedef** használatával rendeljünk szinonim neveket az **int**, **double** és a **char** típusokhoz! A deklarált változóknak adjunk kezdőértéket, és írassuk ki a tartalmukat! (*SajatTipus*)

```
#include <iostream>
using namespace std;
typedef int egesz;
typedef double valos;
typedef char karakter;
int main()
{
  egesz j = 11;
  valos d = 2.3;
  karakter c = '*';
  cout << "j: " << j << endl;
  cout << "d: " << d << endl;
  cout << "c: " << c << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
j: 11
d: 2.3
c: *
```

2.1.4 Konstansok a C++ nyelvben

A **const** használatával definiáljunk konstansokat, adjunk nekik értéket, melyet felhasználunk a programban! Szolgáljon egy *faktor* egész konstans az *adat* szorzására, valamint egy *normal* valós konstans az eredmény osztására! (*Konstans*)

Megjegyzés:

- A konstans tárolók csak olvashatók a programban, értéküket közvetlenül nem lehet megváltoztatni.
- A program végén két `cin.get();` utasítást kell használnunk, mivel olvasáskor az adat utáni *Entert* az első `cin.get();` „lenyeli”, így csak a második hívás függeszti fel a programot, várva egy karakter leütésére.

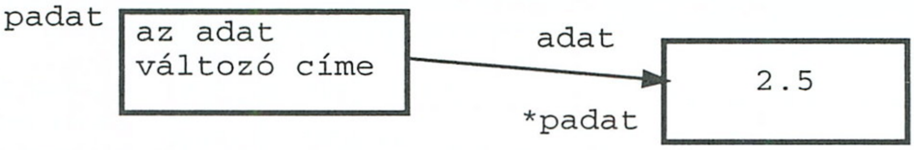
```
#include <iostream>
using namespace std;
const double normal = 1.5;
const int faktor = 8;
int main()
{
  double adat, eredmeny;
  cout << "adat: "; cin >> adat;
  eredmeny = adat * faktor/ normal;
  cout << "eredmeny: " << eredmeny << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
adat: 2.3
eredmeny: 12.2667
```

2.2 Értékek, címek, mutatók

Írjunk programot, amelyben **double** típusú *adat* címét átveszi a **double*** típusú *padat* mutató! Adjunk értéket a mutatónak, majd jelenítsük meg a címeket és az adatokat. Változtassuk meg az *adat* változó értékét, figyeljük meg mi történik! (*MutRef*)



adat	változó tartalma : 2.5	*&adat*	az *adat* változó memóriabeli címe (& - referencia - változó címe)
padat	tartalma cím, amely most megegyezik az adat címével	**padat*	2.5 (a *padat* által mutatott változó tartalma.)

```
#include <iostream>
using namespace std;
int main()
{
  double adat, *padat;
  padat = &adat;
  *padat = 2.5;
  cout << "A mutato letezo objektumra mutat\n";
  cout << "adat valtozo cime (&adat)  : " << &adat << endl;
  cout << "padat mutato tartalma (cim): " << padat << endl;
  cout << "*padat(a padat altal mutatott) valtozo tartalma: "
       << *padat << endl;
  cout << "adat: " << adat << endl;
  cout << "\nadat uj erteke: "; cin >> adat;
  cout << "*padat uj erteke: " << *padat << "  "
       <<"es az adat is: " << adat << endl;
  cin.get(); cin.get();
  return 0;
}
```

Megjegyzés: Mivel az *adat* és a **padat* ugyanarra a memória-területre hivatkozik, akár a **padat*, akár az *adat* tartalmát változtatjuk, *mindkettő értéke azonos lesz*! Értékeljük ki a futtatási eredményt!

A program futásának eredménye:

```
A mutato letezo objektumra mutat
adat valtozo cime (&adat)  : 0x74fdf0
padat mutato tartalma (cim): 0x74fdf0
*padat(a padat altal mutatott) valtozo tartalma: 2.5
adat: 2.5

adat uj erteke: 4.3
*padat uj erteke: 4.3  es az adat is: 4.3
```

2.3 A void* típusú általános mutató

Írjunk programot, amelyben **void*** típusú általános mutatóval végzünk értékadást, és számolást! (*VoidTip*)

Megjegyzés: A C++ nyelv típus nélküli, ún. általános mutatók használatát is lehetővé teszi, amely azonban sohasem jelöl ki tárolót. Ha ilyen mutatót szeretnénk alkalmazni, akkor egy változó memóriabeli címét kell hozzárendelni. A hivatkozott változónak csak akkor adhatunk értéket, vagy végezhetünk vele műveletet, ha felhasználói típuskonverzióval (*cast*) típust rendelünk a cím mellé.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  double w = 2.5, negyzete;
  void* p = &w;
  cout << "w : " << w << endl;
  cout << "*(double *)p: " << *(double *)p << endl;
  negyzete = pow(*(double *)p ,2);
  cout << "negyzete: " <<negyzete << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
w : 2.5
*(double *)p: 2.5
negyzete: 6.25
```

2.4 Többszörös indirektségű mutatók

Írjunk programot, amelyben többszörös indirektségű mutatót használunk, és műveletet végzünk vele! (*TMutato*)

```
#include <iostream>
using namespace std;
int main()
{
  double w, osszeg; // valós változók
  double *pw;       // pw valós (double)típusú mutató, azaz
                    // double változóra mutathat
  double **ppw;     // ppw egy double* típusú mutató, amely
                    // double* változóra, vagyis valósra mutató
                    // pointerre mutat
  w = 1.5;
  pw = &w;
  ppw = &pw;
  osszeg = w + *pw + **ppw;
  cout << "w     : " << w << endl;
  cout << "*pw   : " << w << endl;
  cout << "**ppw : " << w << endl;
  cout << "osszeg: " << osszeg << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
w     : 1.5
*pw   : 1.5
**ppw : 1.5
osszeg: 4.5
```

Megjegyzés: Mivel a címek azonos helyre mutatnak, az összeg a *w* változó tartalmának háromszorosa lesz.

3. Operátorok és kifejezések

A C++ nyelvben a pontosvesszővel lezárt kifejezések – utasítások. A kifejezés egyetlen operandus, vagy operandusok és műveleti jelek (operátorok) kombinációjából épül fel. Az operandusok a C++ nyelv azon elemei, amelyeken az operátorok fejtik ki a hatásukat.

3.1 Aritmetikai operátorok

Írjunk programot, amely bemutatja az értékadás szabályait! (*Ertekado*)

Megjegyzés:

- `y = ( x = 4)/2*3;` kifejezésben az *x* és az *y* is értéket kap.
- A jobbról-balra szabály lehetővé teszi több változó ugyanazon értékre való állítását: `d1 = d2 = 5;`
- Ha egy változó a jobb és a bal oldalon is szerepel, akkor a kifejezés tömörebben is felírható úgy, hogy a műveleti jelet az egyenlőség elé tesszük (üres hely nélkül).
  `z = z/(2+w)` → `z /= 2+w;`
  Lehet még: `+=, -=, *=, %=, &=, |=, ^=, <<=, >>=`

```
#include <iostream>
using namespace std;
int main()
{
  double x, y, z = 10, w = 3, d1, d2;
  y = ( x = 4)/2 * 3;
  cout << "x: " << x << "  y: " << y << endl;
  d1 = d2 = 5; // jobbrol balra szabaly
  cout << "d1: " << d1 << " d2: " << d2 << endl;
  cout << "z: " << z << " w: " << w << endl;
  z /= 2 + w; // azonos a z = z/(2 + w) kifejezéssel
  cout << "z muvelet utani eredmenye: " << z << endl;
  d1 += 2 * w - 1; // azonos a d1 = d1 + (2 * w - 1) kifejezéssel
  cout << "d1 muvelet utani eredmenye: " << d1 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
x: 4  y: 6
d1: 5 d2: 5
z: 10 w: 3
z muvelet utani eredmenye: 2
d1 muvelet utani eredmenye: 10
```

3.2 Léptető operátorok

Írjunk programot, amely bemutatja több egész típusú változó léptetését! (*Lepteto*)

Megjegyzés:

- Egy numerikus változó eggyel való növelése, illetve csökkentése helyett érdemes a léptető operátort használni:
  `i = i+1;` helyett `i++;` vagy `++i;`
  `i = i-1;` helyett `i--;` vagy `--i;`
  `++i; --i;` prefixes alakok.
  `i++; i--;` postfixes alakok.
- Vigyázzunk, ha a prefixes operátort kifejezésben használjuk, akkor a léptetés a kifejezés kiértékelése előtt megy végbe.

```
int i = 2, j = 6, k;
k = ++i - --j;   // i 3, j 5 és k -2 lesz.
```

```
#include <iostream>
using namespace std;
int main()
{
  int i = 5, j = 3, ered1, ered2, ered3, ered4;
  cout << "i= " << i << " j= " << j << endl;
  ered1 = i++ + ++j;
  cout << "Muveletkor j: " << j << " ered1: " << ered1 << endl;
  cout << endl << "Muvelet utan" << endl;
  cout << "i= " << i << " j= " << j << endl;
  ered2 = ++i + j++;
  cout << "Muveletkor i: " << i << " ered2: " << ered2 << endl;
  cout << endl << "Muvelet utan" << endl;
  cout << "i= " << i << " j= " << j << endl;
  ered3 = i-- + j--;
  cout << "ered3: " << ered3 << endl;
  cout << endl << "Muvelet utan" << endl;
  cout << "i= " << i << " j= " << j << endl;
  ered4 = --i + --j;
  cout << "Muveletkor i: " << i << " j: " << j << " ered4: "
       << ered4 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
i= 5 j= 3
Muveletkor j: 4 ered1: 9

Muvelet utan
i= 6 j= 4
Muveletkor i: 7 ered2: 11
```

```
Muvelet utan
i= 7 j= 5
ered3: 12

Muvelet utan
i= 6 j= 4
Muveletkor i: 5 j: 3 ered4: 8
```

3.3 Bitműveletek

Írjunk programot, amelyben **short int** típusú hexadecimális értéket tartalmazó változókon végzünk *vagy* (|), *kizáró vagy* (^), *és* (&) valamint *tagadás*(~) logikai műveleteket! (*Bitmuv1*)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  short int i = 0x1465, j1 = 0x2412, vagy;
  vagy = i | j1;
  cout << hex << i << " | " << j1 << " = " << vagy << endl;
  short int j2 = 0x0077, es;
  es = i & j2;
  cout << endl << hex << i << " & " << j2
       << " = " << es << endl;
  short int j3 =0x14F0, kizarovagy;
  kizarovagy = i ^ j3;
  cout << endl << hex<< i << " ^ " << j3 << " = " << kizarovagy << endl;
  short int negalas;
  negalas = ~i;
  cout << endl << hex << " ~ " << i << " = " << negalas << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1465 | 2412 = 3477

1465 & 77 = 65

1465 ^ 14f0 = 95

~ 1465 = ffffeb9a
```

```
   0001 0100 0110 0101
 | 0010 0100 0001 0010
   0011 0100 0111 0111  →  3477

   0001 0100 0110 0101
 & 0000 0000 0111 0111
   0000 0000 0110 0101  →  65

   0001 0100 0110 0101
 ^ 0001 0100 1111 0000
   0000 0000 1001 0101  →  95

   0001 0100 0110 0101
 ~ 1110 1011 1001 1010  →  eb96

 az előző 4 bájt ffff
```

Írjunk programot, amelyben **short int** típusú hexadecimális értékeken bemutatja a *kizáró vagy* művelettel való kódolást és a kódolt adat visszaállítását! (*Bitmuv2*)

```
#include <iostream>
using namespace std;
int main()
{
  short int kodolo = 0x01542, adat = 0x2f0a;
  cout << hex << "Adat: " << adat << " ^ "
       << "kodolo: " << kodolo << endl;
  adat = adat ^ kodolo;
  cout << hex << "Kodolt adat: " << adat << endl;
  adat = adat ^ kodolo;
  cout << hex << "Visszakodolas (^): " << adat << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 2f0a ^ kodolo: 1542
Kodolt adat: 3a48
Visszakodolas (^): 2f0a
```

3.3.1 Biteltoló műveletek

Írjunk programot, amely egy **short int** típusú hexadecimális értéket kettővel jobbra, illetve balra léptet! (*Biteltolo*)

Megjegyzés:

- Egész szám bitjeinek *n* lépéssel történő balra tolása a szám 2^n értékkel való szorzását eredményezi.
- Egész szám bitjeinek *n* lépéssel történő jobbra tolása a szám 2^n értékkel elvégzett egész osztásának felel meg.

```
#include <iostream>
using namespace std;
int main()
{
  short int a = 0x0065, ered1, ered2;
  ered1 = a >> 2;
  cout << hex << a << " >> 2 = " << ered1 << endl;
  ered2 = a << 2;
  cout << hex <<  a << " << 2 = " << ered2;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
65 >> 2 = 19      0000 0000 0110 0101 >> 2  0000 0000 0001 1001 -> 19
65 << 2 = 194     0000 0000 0110 0101 << 2  0000 0001 1001 0100 ->194
```

3.4 A sizeof operátor

Írjunk programot, amely különböző típusú változóknak kiíratjuk a bájtban megadott méretét a **sizeof** operátor használatával! (*Meret*)

Megjegyzés:

- A **sizeof** művelet kétféleképpen alkalmazható:
  ```
  meret = sizeof  változó; // a változó nevével
  meret = sizeof (int);    // a változó típusával, zárójelben.
  ```

```
#include <iostream>
using namespace std;
int main()
{
  int i;
  short int j;
  long int k;
  float x;
  double y;
  long double w;
  int m1, m2, m3, m4, m5, m6;
  m1 = sizeof(int);
  m2 = sizeof j;
  m3 = sizeof(long int);
  m4 = sizeof(float);
  m5 = sizeof(double);
  m6 = sizeof w;
  cout << "int         merete: " << m1 << endl;
  cout << "short int   merete: " << m2 << endl;
  cout << "long int    merete: " << m3 << endl;
  cout << "float       merete: " << m4 << endl;
  cout << "double      merete: " << m5 << endl;
  cout << "long double merete: " << m6 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
int         merete: 4
short int   merete: 2
long int    merete: 4
float       merete: 4
double      merete: 8
long double merete: 12
```

Megjegyzés: a **long double** mérete implementációfüggő!

3.5 A vessző operátor

Írjunk programot, amely értékadásnál, illetve műveletben használja a vessző operátort! (*Vesszo*)

Megjegyzés:

- A vessző operátort tartalmazó kifejezés balról-jobbra haladva értékelődik ki, a kifejezés értéke és típusa megegyezik a jobb oldali operandus értékével, illetve típusával.
- A programban `m2 = i * (m1 = 4, m1 + 3);` kifejezésben az *m1* először megkapja a 4 értéket, majd utána történik a 3 hozzáadása, és végül az *i* értékével (2) lesz szorozva. Az *m2* értéke 14 lesz.

```
#include <iostream>
using namespace std;
int main()
{
 int i = 1, j, k, n, m, m1,m2;
  n = m = 0;    // jobbról-balra szabály
  i = 2, j =3; // a vessző operátor használata
  m2 = i * (m1 = 4, m1 + 3); // a vessző operátor használata
  cout << "m1 erteke : " << m1 << endl;
  cout << "m2 muvelet: " << m2 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
m1 erteke : 4
m2 muvelet: 14
```

3.6 A fetételes operátor

Írjunk programot, amelyben feltételes operátort használunk két változó közül a legnagyobb értékű kiválasztására! (*Feltelop*)

Megjegyzés:

- A kifejezés operátor (?:) három operandussal rendelkezik: *kif1 ? kif2 : kif3*
- Ha a *kif1* nem 0 (igaz), akkor a *kif2* értéke adja a feltételes kifejezés értékét, ellenkező esetben a kettőspont (:) után lévő *kif3*.
- Az eredményhez kétféleképpen is hozzáférhetünk (bemutatja a példa).

```
#include <iostream>
using namespace std;
int main()
{
  int x = 2, y = 4, max1, max2;
  x > y ? max1 = x : max1 = y;
  max2 = x > y ? x : y;
  cout << "Legnagyobb ertek (max1): " << max1 << endl;
  cout << "Legnagyobb ertek (max2): " << max2 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Legnagyobb ertek (max1): 4
Legnagyobb ertek (max2): 4
```

3.7 Az érvényességi kör (hatókör) operátor

Írjunk programot, amelyben hatókör operátort használunk globális változók elérésére, amennyiben annak neve megegyezik valamely lokális változóéval! (*Hatokor*)

Megjegyzés:

- A :: (hatókör operátorral) a program tetszőleges blokkjából hivatkozhatunk a fájl színtű globális változókra.
- Az osztály adattagjaira is a :: (hatókör operátorral) hivatkozhatunk (lásd később).
- A névterületen definiált nevekre is hivatkozhatunk a :: (hatókör operátorral), ha a **using namespace** deklaráció nélkül kívánjuk elérni azokat (ugyancsak későbbi téma).

```
#include <iostream>
using namespace std;
//globális deklaráció
double a = 1.5;
int j = 12;
int main()
{
  double a = 2.3;
  int j = 5;
  double ered1, ered2, ered3;
  cout << "Lokalis valtozok ertekei:" << endl;
  cout << "a  " << a << endl;
  cout << "j  " << j << endl;
  ered1 = a + j;
  cout << "a + j = " << ered1 << endl << endl;
  cout << "Globalis valtozok ertekei:" << endl;
  cout << "::a  " << ::a << endl;
  cout << "::j  " << ::j << endl;
  ered2 = ::a + ::j;
  cout << "::a + ::j = " << ered2 << endl << endl;
  ::a = 4.2;
```

```
    ::j = -4;
    cout << "Globalis valtozok megvaltozott ertekei:" << endl;
    cout << "::a  " << ::a << endl;
    cout << "::j  " << ::j << endl;
    ered3 = ::a + ::j;
    cout << "::a + ::j = " << ered3 << endl;
    cin.get();
    return 0;
}
```

A program futásának eredménye:

```
Lokalis valtozok ertekei:
a  2.3
j  5
a + j = 7.3

Globalis valtozok ertekei:
::a  1.5
::j  12
::a + ::j = 13.5

Globalis valtozok megvaltozott ertekei:
::a  4.2
::j  -4
::a + ::j = 0.2
```

3.8 Statikus és dinamikus helyfoglalású változók használata

Írjunk programot, amelyben egész típusú, statikus helyfoglalású változókon végzünk műveleteket! Mit tapasztalunk? (*Muvelet1*)

Megjegyzés:

- Két egész szám hányadosa egész, még akkor is, ha a bal oldali változó valós. Ha nem akarunk értékcsökkenést, akkor a számlálót, vagy a nevezőt valóssá kell alakítanunk.
- Vagy az egész számot tizedesponttal, vagy a változót **double** típusmódosítással átalakítjuk valóssá, hogy az osztás művelet elvégzésekor helyes eredményt kapjunk:
  ```
  atlag2 = (a+b)/2.;
  atlag3 = (double)(a+b)/2;
  ```
- Változó beolvasása a ***cin*** adatfolyam használatával a >> operátor segítségével történik. A ***cout*** adatfolyammal rákérdezünk az adatra, és ***cin*** segítségével olvassuk be: `cout << "a: "; cin >> a;`

```
#include <iostream>
using namespace std;
int main()
{ int a, b, c;
  double osszeg, hanyados, atlag1, atlag2, atlag3;
  cout << "a: "; cin >> a;
  cout << "b: "; cin >> b;
  osszeg = a + b;
  cout << "osszeg: " << osszeg << endl;
  c = a - b;
  cout << "kulonbseg: " << c << endl;
  hanyados = a/b;
  cout << "hanyados: " << hanyados << endl;
  atlag1 = (a+b)/2;
  atlag2 = (a+b)/2.;
  atlag3 = (double)(a+b)/2;
  cout << "atlag1: " << atlag1 << endl;
  cout << "atlag2: " << atlag2 << endl;
  cout << "atlag3: " << atlag3 << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a: 5
b: 2
osszeg: 7
kulonbseg: 3
hanyados: 2
atlag1: 3
atlag2: 3.5
atlag3: 3.5
```

3.8.1 A new és a delete operátorok használata

Írjunk programot, amelyben valós változókon végzünk műveleteket, az eredményeket mutatók segítségével tároljuk! (*Muvelet2*)

Megjegyzés:

- A dinamikus változónak a **new** operátorral foglalunk helyet a memóriában, megadva annak típusát: `*szk = new double;`
- A **delete** operátor felszabadítja a **new** operátor által lefoglalt területet: `delete szk;`

```
#include <iostream>
#include <cmath> // sqrt() matematikai függvény eléréséhez szükséges
using namespace std;
int main()
{ double adat1, adat2, *szk, *mk;
  cout << "adat1: "; cin >> adat1;
  cout << "adat2: "; cin >> adat2;
  szk = new double;  // memória-foglalás
  *szk = (adat1 + adat2)/2;
  cout << "Szamtani kozep: " << *szk << endl;
  mk = new double; // memória-foglalás
  *mk = sqrt(adat1 * adat2);
  cout << "Mertani kozep : " << *mk << endl;
  delete szk; // a lefoglalt memória felszabadítása
  delete mk;  // a lefoglalt memória felszabadítása
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
adat1: 1
adat2: 2
Szamtani kozep: 1.5
Mertani kozep : 1.41421

adat1: 1.5
adat2: 2.5
Szamtani kozep: 2
Mertani kozep : 1.93649
```

3.9 Konstansok használata

Írjunk programot, amelyben a kör kerületének és területének számítása során a π értékét konstansként használjuk, valamint a kiírandó szöveget mutató segítségével tároljuk! Használjuk a *cmath* fejlécállományt a ***pow()*** matematikai függvény eléréséhez! (*Konstansok*)

```
#include <iostream>
#include <cmath>
using namespace std;
char *szoveg1 = "kor terulete: ";
char *szoveg2 = "kor kerulete: ";
const double PI = 3.141592654;
int main()
{
  double sugar, korter, korker;
  cout << "Kor sugara: "; cin >> sugar;
  korter = pow(sugar, 2)* PI;
  korker = 2 * sugar * PI;
  cout << szoveg1 << korter << endl;
  cout << szoveg2 << korker << endl << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
sugar: 1
kor terulete: 3.14159
kor kerulete: 6.28319
```

3.10 Saját típus előállítása

Írjunk programot, amelyben a változókat saját típussal definiáljuk. Olvassuk be a gömb sugarát, számítsuk ki a gömb felszínét és térfogatát, és a számításához definiáljuk a π értékét konstansként! A hatványozáshoz használjuk a ***pow()*** matematikai függvényt a ***cmath*** fejlécállományból! (*Typedef*)

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.141592654;
typedef int egesz;
typedef double valos;
int main()
{
  egesz gombsugar;
  valos sugar, gombfelsz, gombterf;
  cout << "gomb sugara: "; cin >> gombsugar;
  gombfelsz = 4 * pow(gombsugar, 3) * PI/3;
  gombterf = 4 * pow(gombsugar, 2) * PI;
  cout << "gomb felszine : " << gombfelsz << endl;
  cout << "gomb terfogata: " << gombterf << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
gomb sugara: 1
gomb felszine : 4.18879
gomb terfogata: 12.5664
```

3.11 Szabványos I/O használata

Írjunk programot, amelyben beolvassuk a téglalap két oldalának hosszát és kiszámítjuk a téglalap területét és kerületét, majd kiírjuk az eredményeket (alapértelmezett kiírás)! (*Kiir1*)

```
#include <iostream>
using namespace std;
int main()
{
  double a, b, ter, ker;
  cout << "a oldal: "; cin >> a;
  cout << "b oldal: "; cin >> b;
  ter = a*b;
  ker = 2*(a+b);
  cout << "Teglalap terulete: " << ter << endl;
  cout << "Teglalap kerulete: " << ker << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a oldal: 3.56
b oldal: 4.28
Teglalap terulete: 15.2368
Teglalap kerulete: 15.68
```

Írjunk programot, amelyben beolvassuk a téglalap két oldalának hosszát, majd kiszámítjuk a téglalap területét és kerületét! Az eredményeket fixpontosan jelenítsük meg! (*Kiir2*)

```
#include <iostream>
using namespace std;
int main()
{
  double a, b, ter, ker;
  cout.setf(ios::fixed);  // fixpontos kiírási mód beállítása
  cout << "a oldal: "; cin >> a;
  cout << "b oldal: "; cin >> b;
  ter = a*b; ker = 2*(a+b);
  cout << "Teglalap terulete: " << ter << endl;
  cout << "Teglalap kerulete: " << ker << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a oldal: 1200.5
b oldal: 542.3
Teglalap terulete: 651031.150000
Teglalap kerulete: 3485.600000
```

Írjunk programot, amelyben beolvassuk a téglalap két oldalának hosszát, és kiszámítjuk a téglalap területét és kerületét! Írjuk ki az eredményeket lebegőpontosan! (*Kiir3*)

```
#include <iostream>
using namespace std;
int main()
{
  double a, b, ter, ker;
  cout.setf(ios::scientific);  // lebegőpontos megadási mód
  cout << "a oldal: "; cin >> a;
  cout << "b oldal: "; cin >> b;
  ter = a*b; ker = 2*(a+b);
  cout << "Teglalap terulete: " << ter << endl;
  cout << "Teglalap kerulete: " << ker << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a oldal: 2.5
b oldal: 3.2
Teglalap terulete: 8.000000e+00
Teglalap kerulete: 1.140000e+01
```

Írjunk programot, amelyben beolvassuk a téglalap két oldalának hosszát, és kiszámítjuk a téglalap területét és kerületét! Jelenítsük meg az eredményeket 4 értékes jegyre kerekítve! (*Kiir4*)

Megjegyzés:

- Az értékes jegyek számának (*n*) beállítása:
  `cout.precision(n);`
- Kiírja a pozitív előjelet is:
  `cout.setf(ios::showpos);`
- Megszünteti a pozitív előjel kiírásának kérését:
  `cout.unsetf(ios::showpos);`

```
#include <iostream>
using namespace std;
int main()
{
  double a, b, ter, ker;
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.precision(4);
  cout << "a oldal: "; cin >> a;
  cout << "b oldal: "; cin >> b;
  ter = a*b;
  ker = 2*(a+b);
  cout.setf(ios::showpos);
  cout << "Teglalap terulete: " << ter << endl;
  cout.unsetf(ios::showpos);
  cout << "Teglalap kerulete: " << ker << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a oldal: 2.5
b oldal: 4.2
Teglalap terulete: +10.5000
Teglalap kerulete: 13.4000
```

Írjunk programot, amelyben három egész változó tartalmát írjuk ki 4, 6, 6 mezőszélességgel! Használjuk a ***setw()*** manipulátort! (*Kiir5*)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  int i = 12, j = -156, k = 1243;
  cout << "1234567890123456" << endl;
  cout << setw(4) << i << setw(6) << j
       << setw(6) << k << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1234567890123456
  12  -156  1243
```

Írjunk programot, amelyben egy egész változó tartalmát írjuk ki decimális, hexadecimális és oktális számrendszerben! (*Kiir6*)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  int szam = 10;
  cout << dec << "decimalis     : " << szam << endl;
  cout << hex << "hexadecimalis : " << szam << endl;
  cout << oct << "oktalis       : " << szam << endl;
  szam++; cout << endl;
  cout << "Megvaltozott ertek (marad az oct beallitas) : "
       << szam << endl;
  cout << dec << "Megvaltozott ertek (dec visszavaltas)       : "
       << szam << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
decimalis     : 10
hexadecimalis : a
oktalis       : 12

Megvaltozott ertek (marad az oct beallitas) : 13
Megvaltozott ertek (dec visszavaltas)       : 11
```

Írjunk programot, amelyben használjuk a jobbra, illetve a balra igazítást, a váltást pedig **long** típusú *ios::adjustfield* **long** konstans segítségével végezzük! (*Kiir7*)

Megjegyzés:

- Adott szélességben, az adat előtt vagy mögött a megadott karakterrel kitölthetjük a sort. A ***with***(*n*) *n* paramétere határozza meg a szélességet. A ***fill***() függvény a paraméterében megadott karakterrel tölti ki a maradék helyet, a meghatározott szélességben (az *iomanip* fejlécállomány használata szükséges).

  `cout.width(12); cout.fill('?');`

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  int szam = 10;
  cout.fill('!'); cout.width(10);
  cout << szam << endl;
  cout.width(12);
  cout.fill('?');
  cout.setf(ios::left,ios::adjustfield); // balra igazítás
  cout << szam << endl;
  szam++;
  cout.width(12); cout.fill('*');
  cout << szam << endl;
  szam++;
  cout.width(12); cout.fill('&');
  cout.setf(ios::right);  // jobbra igazítás
  cout << szam << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
!!!!!!!!10
10??????????
11**********
&&&&&&&&&&12
```

Írjunk programot, amelyben karaktereket olvasunk a ***cin*** adatfolyammal, és más sorrendben írjuk vissza! (*Olvas1*)

```
#include <iostream>
using namespace std;
int main()
{
  char c1, c2, c3, c4,c5;
  cout << "\nSzo (max.5 karakter): ";
  cin.get(c1); cin.get(c2); cin.get(c3);
  cin.get(c4); cin.get(c5);
  cout<<"\nBeolvasott szo         : "<< c1 << c2 << c3 <<c4 <<c5 <<endl;
  cout << "Mas sorrendben kiirva: ";
  cout.put(c4); cout.put(c3); cout.put(c1);
  cout.put(c2); cout.put(c5);
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Szo (max.5 karakter): csoki

Beolvasott szo       : csoki
Mas sorrendben kiirva: kocsi
```

4. A C++ nyelv utasításai

Ebben a fejezetben a szelekciós, vezérlésátadó és iterációs utasításokra mutatunk be különféle feladatokat.

4.1 Utasítások és blokkok

Tetszőleges kifejezés utasítás lesz, ha pontosvesszőt (;) helyezünk mögé:

```
j++;
```

Üres utasítás egyetlenegy pontosvesszőből áll és utasítást helyettesít, amennyiben mást nem akarunk végrehajtani azon a helyen.

Több utasítást egy utasítássá vonhatunk össze a kapcsos zárójelekkel `{ }`. A szelekciós és iterációs utasítások alaphelyzetben csupán egy tevékenységet (utasítás) hajtanak végre. Azonban kapcsos zárójelekkel összefogott utasítások egy utasításnak látszanak számukra. Ha elfelejtjük blokkba tenni a megfelelő utasításokat, akkor csak az elsőt futtatják.

4.2 Az if utasítás

Feltételes utasításnak két formája van.

- Az **if** utasítás:

  if *(kifejezés)*
  utasítás

  Ha a *kifejezés* nem nulla, akkor a feltétel igaz, így az *utasítás* lesz végrehajtva.

- Az **if-else** utasításnál, ha a feltétel teljesül az *utasítás1* kerül végrehajtásra, különben az *utasítás2*.

  if *(kifejezés)*
  utasítás1
  else
  utasítás2

Írjunk programot, amely beolvas egy egész számot, és kijelzi, ha negatív, illetve a 10-hez való viszonyát! (*Feltetel_01*)

```
#include <iostream>
using namespace std;
```

```
int main()
{
  int x;
  cout << "Egesz szam: "; cin >> x;
  if ( x < 0 ) cout << " negativ, kisebb 0-nal " << endl;
  else if (x < 10)
   cout << " kisebb 10-nel" << endl;
  else
   cout << " nem kisebb 10-nel " << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Egesz szam: 13
 nem kisebb 10-nel
```

Írjunk programot, amely három valós pozitív számnak kiszámítja a mértani közepét, hibát jelez, ha a kívánt (pozitív) feltétel nem teljesül! (*Feltetel_02*)

```
#include <iostream>
using namespace std;
#include <cmath>
int main()
{
  double x1, x2, x3, mk;
  cout << "Harom pozitiv szam mertani kozepe" << endl;
  cout << "1. adat : "; cin >> x1;
  cout << "2. adat : "; cin >> x2;
  cout << "3. adat : "; cin >> x3;
  if (x1 > 0 && x2 > 0 && x3 > 0)
  {
    mk = pow(x1*x2*x3,1./3);
    cout << "Mertani kozep: " << mk << endl;
  }
  else cerr << "Hibas adat! " << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
Harom pozitiv szam mertani kozepe
1. adat : 1
2. adat : 2
3. adat : 3
Mertani kozep: 1.81712

Harom pozitiv szam mertani kozepe
1. adat : 1
2. adat : 2
3. adat : -4
Hibas adat!
```

Írjunk programot, amely beolvas egy pozitív egész számot, és kiírja: páros, vagy osztható 3-mal, illetve 3, és 5-tel osztható, vagy egyikkel sem osztható! (*Feltetel_03*)

```
#include <iostream>
using namespace std;
int main()
{
  int szam, db = 0;
  cout << "Pozitiv szam: "; cin >> szam;
  if (szam <= 0) cerr << "Hibas adat!" << endl;
  else
  {
   if (szam % 2 == 0)
   {
     cout << "oszthato 2-vel, paros szam" << endl;
     db++;
   }
   if (szam %3 == 0)
   {
     cout << "oszthato 3-mal" << endl;
     db++;
   }
   if (szam %3 == 0 && szam % 5 == 0)
   {
     cout << "oszthato 3-mal es 5-tel is" << endl;
     db++;
   }
   if (!db) cout << "nem oszthato 2, 3 es 5-tel" << endl;
  }
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
Pozitiv szam: 12
oszthato 2-vel, paros szam
oszthato 3-mal

Pozitiv szam: 13
nem oszthato 2, 3 es 5-tel
```

Írjunk programot, amely beolvas három valós értéket: 2 darab zárthelyi és a házi feladat eredményét! Ellenőrizzük az adatokat, majd az átlagukat kiszámítva, szöveges formában adjunk jegyet! (*Feltetel_04*)

```
#include <iostream>
using namespace std;
int main()
{
  double zh1, zh2, hz, atlag;
  cout << "1. zarthelyi eredmenye: "; cin >> zh1;
  cout << "2. zarthelyi eredmenye: "; cin >> zh2;
  cout << "Hazi feladat eredmenye: "; cin >> hz;
  if (zh1 > 0 && zh1 <=5 && zh2 > 0 && zh2 <=5 && hz >0 && hz <=5)
  {
    cout << endl << "Felevi jegy: ";
    atlag = (zh1 + zh2 + hz)/3; cout << atlag;
    if (atlag < 1.5 ) cout << " elegtelen " << endl;
    if (1.5 <= atlag && atlag < 2.5 )
       cout << " elegseges " << endl;
    if (2.5 <= atlag && atlag < 3.5 )
       cout << " kozepes " << endl;
    if (3.5 <= atlag && atlag < 4.5 )
       cout << " jo " << endl;
    if (4.5 <= atlag && atlag <=5 )
       cout << " jeles " << endl;
  }
  else cerr << "Hibas adat!" << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. zarthelyi eredmenye: 2.5
2. zarthelyi eredmenye: 3.2
Hazi feladat eredmenye: 4.7

Felevi jegy: 3.46667 kozepes
```

Írjunk programot, amely beolvas egy valós számot! Ha az adat 0 vagy negatív, akkor emeljük négyzetre, ha pozitív vonjunk belőle gyököt! (*Feltetel_05*)

```
#include <iostream>
using namespace std;
#include <cmath>
int main()
{
  double adat, eredmeny;
  cout << "Adat : "; cin >> adat;
  if (adat <= 0) eredmeny = pow(adat,2);
  else eredmeny = pow(adat,1./2);
  cout << "Eredmeny: " << eredmeny << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
Adat : -4
Eredmeny: 16

Adat : 36
Eredmeny: 6
```

Írjunk programot, amely beolvassa a téglalap két oldalának hosszát. Számítsuk ki a téglalap területét és kerületét! Adjunk hibajelzést, ha bármely oldalhossz 0, vagy negatív szám! Ha a programban használjuk a ***system**("PAUSE")* függvényhívást, akkor a program, ha a numerikus adat helyett hibás karaktert (nem számjegy) kapott, akkor is felfüggesztődik! (*Hiba*)

Megjegyzés:

- A c*stdlib* fejlécállomány szükséges a ***system**()* használatához. Ha a `system("PAUSE");` utasítással zárjuk a programot, hibás adat beírásakor is a

  ```
  folytatáshoz nyomjon meg egy billentyűt...
  ```

  üzenettel fejeződik be a program futása.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  double a, b, ter, ker;
  cout << "a oldal: "; cin >> a;
  cout << "b oldal: "; cin >> b;
  if ( a >=0 && b >=0)
  {
    ter = a*b;
    ker = 2*(a+b);
    cout << "Teglalap terulete: " << ter << endl;
    cout << "Teglalap kerulete: " << ker << endl;
  }
  else
    cerr << "Hibas adat!" << endl;
  system("PAUSE");
  return 0;
}
```

Program futásának eredményei:

```
a oldal: 2
b oldal: 3
Teglalap terulete: 6
Teglalap kerulete: 10
 folytatáshoz nyomjon meg egy billentyűt...
```

A b oldal negatív szám esetén:

```
a oldal: 2
b oldal: -3
Hibas adat!
A folytatáshoz nyomjon meg egy billentyűt...
```

Numerikus érték helyett s betűt olvasott:

```
a oldal: s
b oldal: Hibas adat!
A folytatáshoz nyomjon meg egy billentyűt...
```

Írjunk programot, amely bemutatja a rövidzár kiértékelés működését! (*Rovidzar*)

Megjegyzés:

Ha a feltételes utasításban a *vagy* kapcsolat bal oldala teljesül, akkor a jobb oldalán lévő kifejezés már nem kerül kiértékelésre, azaz nem hajtódik végre. Ugyanazt tapasztaljuk, ha az *és* kapcsolat bal oldala hamis.

Például, ha a feltételben az *i* nem nulla, akkor csökkent az *i* tartalma, de a *j* tartalma nem fog növekedni (lásd a program futásának eredményét).

```
if (i-- || j++)
     k = i + j;
```

```
#include <iostream>
using namespace std;
int main()
{
  int i = 3, j = 0, k = 0;
  cout << "if vegrehajtasa elott" << endl;
  cout << "i: " << i << " j: " << j << endl;
  if (i-- || j++)
     k = i + j;
  cout << "if vegrehajtasa utan" << endl;
  cout << "i: " << i << " j: " << j << endl;
  cout << "k: " << k << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
if vegrehajtasa elott
i: 3 j: 0
if vegrehajtasa utan
i: 2 j: 0
k: 2
```

4.3 A switch utasítás

A **switch** típusú elágaztatásban a vezérlés egy *kifejezés* értékétől függően több lehetséges utasítás valamelyikére kerül. Az utasítás általános alakja:

```
switch (kifejezés)
{
  case konstans_kifejezés:
              utasítások
  . . .
  default:
              utasítások
}
```

- A **switch** utasítás kiértékeli a *kifejezést,* és a megfelelő **case** címkére adja át a vezérlést, amelyben a *konstans_kifejezés* értéke megegyezik a *kifejezés* értékével. Ha egyikkel sem egyezik meg, akkor a **default** címkével megjelölt utasítással folytatódik a program futása.
- A konstans kifejezés csak egész jellegű (**int**, **enum**, **char**, **bool** stb. lehet).
- Az adott címkéhez tartozó programrészlet végrehajtása után a **goto**, **break** vagy **return** utasítással léphetünk ki a **switch** utasításból.

Írjunk programot, amely beolvas két valós számot és egy karaktert, amely meghatározza, hogy számtani (s,S), vagy mértani közepet (m,M) számolunk! Ellenőrizzük az adatok helyességét, és szövegesen jelenítsük meg az eredményt! (*Switch_01*)

```
#include <iostream>
#include <cmath>
#include <cctype> // a toupper() függvény használatához szükséges
using namespace std;
int main()
{
  double x, y, ered;
  char jel;
  cout << "s, S: szamtani kozep szamitasa" << endl;
  cout << "m, M: mertani  kozep szamitasa" << endl << endl;
  cout << "1. adat: " ; cin >> x;
  cout << "2. adat: "; cin >> y;
  cout << "Melyik muvelet (s,S vagy m,M): ";
  cin >> jel;
  if ( x > 0 && y > 0 && jel == 's' || toupper(jel) == 'S' ||
        jel == 'm' || toupper(jel) == 'M')
  {
    switch (jel)
    {
     case 's':
     case 'S': ered = (x + y)/2;
          cout << endl << "Szamtani kozep = " << ered << endl; break;
     case 'm':
     case 'M': ered = sqrt(x * y);
          cout << endl << "Mertani kozep = " << ered << endl; break;
    }
  }
  else
     cerr << "Hibas adat!" << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
s, S: szamtani kozep szamitasa
m, M: mertani  kozep szamitasa

1. adat: 1
2. adat: 2
```

```
Melyik muvelet (s,S vagy m,M): s

Szamtani kozep = 1.5

s, S: szamtani kozep szamitasa
m, M: mertani  kozep szamitasa

1. adat: 1
2. adat: 2
Melyik muvelet (s,S vagy m,M): M

Mertani kozep = 1.41421
```

Írjunk programot, amely beolvas három valós számot: két zárthelyi és a házi feladat eredményét! A leellenőrzött adatokat kerekítve alakítsuk egész számmá, majd az év-végi eredményt szövegesen jelenítsük meg! (*Switch_02*)

```
#include <iostream>
using namespace std;
int main()
{
  double zh1, zh2, hz, atlag;
  int jegy;
  cout << "1. zarthelyi eredmenye: "; cin >> zh1;
  cout << "2. zarthelyi eredmenye: "; cin >> zh2;
  cout << "Hazi feladat eredmenye: "; cin >> hz;
  if (zh1 > 0 && zh1 <=5 && zh2 > 0 && zh2 <=5 && hz >0 && hz <=5)
  {
    atlag = (zh1 + zh2 + hz)/3;
    jegy = int(atlag + 0.5);
    cout << endl << "Jegyek atlaga : " << atlag << endl;
    cout << "Kerekitett jegy : " << jegy << endl;
    cout << endl << "Felevi jegy: ";
    switch (jegy)
    {
      case 1:  cout << " elegtelen " << endl; break;
      case 2:  cout << " elegseges " << endl; break;
      case 3:  cout << " kozepes " << endl; break;
      case 4:  cout << " jo " << endl; break;
      case 5:  cout << " jeles " << endl; break;
    }
  }
  else cerr << "Hibas adat!" << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
1. zarthelyi eredmenye: 2.5            1. zarthelyi eredmenye: 3.2
2. zarthelyi eredmenye: 3.2            2. zarthelyi eredmenye: 4.3
Hazi feladat eredmenye: 4              Hazi feladat eredmenye: 3.8

Jegyek atlaga : 3.23333                Jegyek atlaga : 3.76667
Kerekitett jegy : 3                    Kerekitett jegy : 4

Felevi jegy:  kozepes                  Felevi jegy:  jo
```

Írjunk programot, amely a beolvasott karaktertől függően különböző típusú adatot képes fogadni! Ha csillagot (*) olvas, a következő adat karakter, ha & karaktert olvas, a várt adat egész szám, # esetén az adat valós szám lehet. (*Switch_03*)

Megjegyzés:

Ebben a feladatban is alkalmazzuk a `system("PAUSE");` hívását a program végén, mert ez hibás adatok esetén is felfüggeszti a program működését.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
  char c1, c2;
  int szam1;
  double szam2;
  cout << "Karakter (*,&,#): "; cin.get(c1);
  switch (c1)
  {
    case '*': cin.get(c2); cout << "Karakter: " << c2; break;
    case '&': cin >> szam1; cout << "Egesz szam: " << szam1; break;
    case '#': cin >> szam2; cout << "Valos szam: " << szam2; break;
    default:
       cerr << "Hibas karakter!" << endl;
  }
  cout << endl;
  system("PAUSE");
  return 0;
}
```

A program futásának eredményei:

```
Karakter (*,&,#): *a
Karakter: a
A folytatáshoz nyomjon meg egy billentyűt...

Karakter (*,&,#): &12
Egesz szam: 12
A folytatáshoz nyomjon meg egy billentyűt...

Karakter (*,&,#): #2.56
Valos szam: 2.56
A folytatáshoz nyomjon meg egy billentyűt...

Karakter (*,&,#): @
Hibas karakter!

A folytatáshoz nyomjon meg egy billentyűt...
```

4.4 Ciklusutasítások

A programozási nyelvekben ciklusnak nevezzük azokat az utasításokat, amelyek bizonyos tevékenysége automatikus ismétlését biztosítják.

4.4.1 A while ciklus

A **while** ciklus mindaddig ismétli a ciklus törzsében lévő utasításokat, míg a vizsgált *kifejezés* (vezérlőfeltétel) értéke igaz (nem 0). Az utasítás általános alakja:

```
while (kifejezés)
{
 // a ciklus törzse
}
```

Írjunk programot, amely beolvassa az adatok darabszámát, majd pedig a megfelelő számú adatot! Számítsuk ki a ciklusban beolvasott adatok összegét és átlagát! (*While_01*)

```
#include <iostream>
using namespace std;
int main()
{
  int i, n;
  double x, atlag, osszeg = 0;
  cout << "Adatok szama: "; cin >> n;
  i = 0;
  while (i < n)
  {
    cout << "Adat: "; cin >> x;
    osszeg += x;
    i++;
  }
  atlag = osszeg/n;
  cout << endl << "Adatok osszege: " << osszeg << endl;
  cout << "Adatok atlaga : " << atlag << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 3
Adat: 1
Adat: 2
Adat: 3

Adatok osszege: 6
Adatok atlaga : 2
```

Írjunk programot, amely megadott darabszámú adatot olvas be! Számítsuk ki a ciklusban beolvasott pozitív adatok átlagát és a negatív adatok összegét! (*While_02*)

```
#include <iostream>
using namespace std;
int main()
{
  int i, n, poz_db = 0;
  double szam, pozatlag = 0, negosszeg = 0;
  cout << "Adatok szama: "; cin >> n;
  i = n;
  while (i > 0)
  {
    cout << "Adat: "; cin >> szam;
    if (szam >= 0)
    {
      pozatlag += szam;
      poz_db++;
    }
    else negosszeg += szam;
    i--;
  }
  pozatlag /= poz_db;
  cout << endl << "Pozitiv adatok atlaga: " << pozatlag << endl;
  cout << "Negativ adatok osszege: " << negosszeg << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 6
Adat: 2
Adat: 3
Adat: -8
Adat: -4
Adat: -10
Adat: 5

Pozitiv adatok atlaga: 3.33333
Negativ adatok osszege: -22
```

Írjunk programot, amely végtelen **while** ciklusban addig olvas pozitív egész számokat, míg egymás után 2 db zérust nem olvasott. A második 0 adatot ne számoljuk bele az adatok darabszámába! Számláljuk meg az adatok darabszámát, és számítsuk ki a beolvasott adatok összegét! (*While_03*)

Megjegyzés:

- Negatív szám beolvasásakor a **continue** utasítás a következő iterációt hajtja végre, a **continue** mögötti utasítások ekkor kimaradnak.
- A nullákat számlálni kell.

- Ha a beolvasott adat nem 0, akkor a *nulla* számláló tartalmát nullázni kell.
- Ha a nullák száma 2, a darabszámláló tartalmát csökkentsük eggyel, majd a **break** utasítással lépjünk ki a végtelen ciklusból!

```
#include <iostream>
using namespace std;
int main()
{
  int i, n = 0, adat, osszeg = 0, nulla = 0;
  while(1)
  {
    cout << "Adat: "; cin >> adat;
    if ( adat < 0) continue;
    osszeg += adat; n++;
    if (adat == 0) nulla++;
    if (nulla == 2) {  n--; break; }
    if (adat != 0) nulla = 0;
  }
  cout << endl << "Adatok szama: " << n << endl;
  cout << "Osszeg: " << osszeg << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 1
Adat: 2
Adat: 0
Adat: 3
Adat: 0
Adat: 0

Adatok szama: 5
Osszeg: 6
```

4.4.2 A for ciklus

A **for** utasítást általában akkor használjuk, ha a ciklus törzsében lévő utasításokat egy számsorozat elemein végigléptetve akarjuk végrehajtani. Az utasítás általános formája:

> **for**(*inicializáló_kif*; *feltétel_kif*; *léptető_kif*)
> {
>   *// a ciklus törzse*
> }

Írjunk programot, amely adatokat olvas be. Számláljuk meg a ciklusban beolvasott pozitív, negatív és a zérus számokat! (*For_01*)

```
#include <iostream>
using namespace std;
int main()
{ int i,n, poz = 0, neg = 0, zero = 0;
  double x;
  cout << "Adatok szama: "; cin >> n;

  for (i = 0; i<n; i++)
  {
    cout << "Adat: " ; cin >> x;
    if ( x < 0 ) neg++;
    else if ( x > 0 ) poz ++;
    else zero++;
  }
  cout << endl << "Pozitiv adatok szama: " << poz << endl;
  cout << "Negativ adatok szama: " << neg << endl;
  cout << "Zerus adatok szama   : " << zero << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 5
Adat: 2
Adat: -3
Adat: 0
Adat: 6
Adat: -4

Pozitiv adatok szama: 2
Negativ adatok szama: 2
Zerus adatok szama   : 1
```

Írjunk programot, amely adott számú adatot olvas be! A ciklusban beolvasott számokat vizsgálva, számoljuk meg a párosakat, páratlanokat és a 3-mal oszthatókat! (*For_02*)

Megjegyzés:
A ciklus utasításban a ciklusváltozót a ciklus fejében is definiálhatjuk, de ekkor az értékét csak a ciklus magjában használhatjuk. A ciklus lefutása után a változó elvész:

```
for(int i = n; i > 0; i--)
{
 . . .
}
// i definíció nem érvényes
```

```
#include <iostream>
using namespace std;
int main()
{
  int n, szam, paros=0, paratlan=0, oszto3 = 0;
  cout << "Adatok szama: "; cin >> n;
  for(int i = n; i > 0; i--)
  {
    cout << "Adat:  "; cin >> szam;
    if ( szam % 2 == 0) paros++;
    else paratlan++;
    if (szam % 3 == 0) oszto3++;
  }
  cout << endl << "Parosak szama: " << paros << endl;
  cout << "Paratlanok szama: " << paratlan << endl;
  cout << "3-mal oszthatok szama: " << oszto3 << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 5
Adat:  2
Adat:  6
Adat:  13
Adat:  4
Adat:  9

Parosak szama: 3
Paratlanok szama: 2
3-mal oszthatok szama: 2
```

Írjunk programot, amely végtelen **for** ciklusban addig olvas pozitív egész számokat, míg egymás után 2 db zérust nem olvasott! A második 0 adatot ne számoljuk bele az adatok darabszámába! Számláljuk meg az adatok darabszámát, és számítsuk ki a beolvasott adatok összegét! (*For_03*)

Megjegyzés:
A végtelen **for** ciklus, amelyből hiányzik a kezdő, a végérték és a lépésköz:

```
for( ; ; )
{  }
```

```
#include <iostream>
using namespace std;
int main()
{
  int i, n = 0, adat, osszeg = 0, nulla = 0;
  for(;;)
  {
    cout << "Adat: "; cin >> adat;
    if (adat < 0) continue;
    osszeg += adat; n++;
    if (adat == 0) nulla++;
    if (nulla == 2) {  n--; break; }
    if (adat != 0) nulla = 0;
  }
  cout << endl << "Adatok szama: " << n << endl;
  cout << "Osszeg: " << osszeg << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adat: 5
Adat: -2
Adat: 3
Adat: 0
Adat: 3
Adat: 0
Adat: 0

Adatok szama: 5
Osszeg: 11
```

4.4.3 A do-while ciklus

A **do-while** ciklus törzse egyszer tesztelés nélkül hajtódik végre. A ciklus addig működik, amíg a *kifejezés* igaz (nem 0). Az utasítás általános alakja:

do
{
// a ciklus törzse
}**while** (*kifejezés*);

Írjunk programot, amely a valós alapot pozitív egész kitevőre emel! (*Dowhile_01*)

```
#include <iostream>
using namespace std;
int main()
{
  double alap, hatvany = 1;
  int kitevo, i;
  cout << "Valos szamot emelunk pozitiv egesz kitevore " << endl;
  cout << "Alap   : "; cin >> alap;
  cout << "Kitevo: "; cin >> kitevo;
  if (kitevo == 0)
   cout << endl << "Hatvany erteke: " << hatvany << endl;
  else
  {
    if ( kitevo > 0)
    {
      i = 1;
      hatvany = alap;
      do
      {
        hatvany *= alap;
        i++;
      }while (i < kitevo);
      cout << endl << "Hatvany erteke: " << hatvany << endl;
   }
   else
   {
     cerr << "A kitevo nem pozitiv szam: " << kitevo << endl;
   }
  }
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
Valos szamot emelunk pozitiv egesz kitevore
Alap  : 2
Kitevo: 0

Hatvany erteke: 1


Valos szamot emelunk pozitiv egesz kitevore
Alap  : 2.5
Kitevo: 2

Hatvany erteke: 6.25

Valos szamot emelunk pozitiv egesz kitevore
Alap  : 3
Kitevo: -2
A kitevo nem pozitiv szam: -2
```

5. Származtatott típusok

Ebben a fejezetben különböző feladatokat mutatunk be egy- és többdimenziós tömbök és struktúrák kezelésére. Használunk statikus és dinamikus adatstruktúrákat is a megoldások összehasonlítására.

5.1 Tömbök, sztringek és mutatók

A tömb azonos típusú adatok halmaza, amelyek a memóriában folytonosan helyezkednek el. A tömb elemeire az elemsorszámmal (index), és az indexelés operátorával (`[ ]`) hivatkozhatunk. A tömbindexek sorszáma mindig 0-val kezdődik.

5.1.1 Egydimenziós tömbök

A egydimenziós tömbök általános alakja:

típus tömbnév[méret];

Írjunk programot, amely beolvassa a tömb aktuális elemszámát! Végezzünk ellenőrzést, ne fogadjunk el rossz elemszámot! A számláljuk meg a nulla adatokat, a pozitív és negatív elemeket pedig adjuk össze, és számítsuk ki az átlagukat! (*Tomb_01*)

```
#include <iostream>
using namespace std;
const int MERET = 20;
int main()
{
  int x[MERET], db = 0, adatdb = 0, osszeg = 0, nulla = 0;
  double atlag;
  do
  {
    cout << "Elemek szama (max. 20): "; cin >> db;
  }while (db <=0 || db > 20);
  for (int i = 0; i<db; i++)
  {
    cout << "Adat: "; cin >> x[i];
    if (x[i] == 0) nulla++;
    else
    {
      osszeg += x[i];
      adatdb++;
    }
  }
  atlag = (double)osszeg/adatdb;
  cout << endl << "Adatok szama: " << adatdb << endl;
  cout << "Osszeg : " << osszeg << endl;
  cout << "Atlag  : " << atlag << endl;
  cout << "Nulla elemek szama: " << nulla << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama (max. 20): 5            Adatok szama: 4
Adat: 11                             Osszeg : 18
Adat: 0                              Atlag  : 4.5
Adat: -3                             Nulla elemek szama: 1
Adat: 4
Adat: 6
```

Írjunk programot, amely *MERET* darabszámú valós adatot olvas be! Csak az *index* tömbben megadott elemeket jelenítsük meg, és számítsuk ki az összegüket! (*Tomb_02*)

Megjegyzés:

- Egy tömb elemszámát meghatározhatjuk a **sizeof** függvénnyel úgy, hogy vesszük a tömb teljes méretét (**sizeof**(*tombnev*)) és osztjuk a *tombnev[0]* elemének a méretével:

  ```
  index_db = sizeof(index)/sizeof(index[0]);
  ```

- A *setw()* használatához az *<iomanip>* fejlécállomány szükséges.

```
#include <iostream>
#include <iomanip>
using namespace std;
int index[] = {1,3,4,7 };
const int index_db = sizeof(index)/sizeof(index[0]);
const int MERET = 10;
int main()
{
  double y[MERET], atlag = 0;
  for (int i = 0; i<MERET; i++)
  {
    cout << setw(2) << i << ". elem: "; cin >> y[i];
  }
  cout << endl << "Osszeadando elemek" << endl;
  cout << "Index" << "  Adat" << endl;
  for (int i = 0; i<index_db; i++)
  {
    atlag += y[index[i]];
    cout << setw(3) << index[i]
         << "      " << y[index[i]] << endl;
  }
  atlag /= index_db;
  cout << endl << "Atlag: " << atlag << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
0. elem: 4                         Osszeadando elemek
1. elem: 1                         Index  Adat
2. elem: 3                           1      1
3. elem: 7                           3      7
4. elem: 6                           4      6
5. elem: 2                           7      8
6. elem: 9
7. elem: 8                         Atlag: 5.5
8. elem: 5
9. elem: 0
```

Írjunk programot, amely beolvassa a tömb aktuális elemszámát, ellenőrizve, hogy ne lépje túl a definiált indexhatárt! Olvassuk be a tömb elemeit, keressük meg a legkisebb adatát a *min* változóba, majd csökkentsük a tömb minden elemét a legkisebb adattal! (*Tomb_03*)

```
#include <iostream>
using namespace std;
#include <iomanip>
typedef int tombtipus[20];
int main()
{
  tombtipus x;
  int db, min;
  do
  {
    cout << "Elemek szama (max. 20): ";
    cin >> db;
  }while ( 0 >= db || db > 20);
  for (int i = 0; i<db; i++)
  {
    cout << i << ". elem: "; cin >> x[i];
  }
  min = x[0]; // a legkisebb adatkeresés kezdőértékének beállítása
  for (int i = 1; i<db; i++)
  {
    if (x[i] < min) min = x[i]; // átírás, ha van kisebb
  }
  cout << endl << "Legkisebb adat: " << min << endl;
  cout << "Legkisebb adattal csokkentett tomb elemei" << endl;
  cout << "Index   Elem " << endl;
  for (int i = 0; i<db; i++)
  {
    x[i] = x[i] - min; // az elemek csökkentése a legkisebb elemmel
    cout << setw(3) << i << setw(9) << x[i] << endl;
  }
  cin.get();cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama (max. 20): 5     Legkisebb adat: 2
0. elem: 5                    Legkisebb adattal csokkentett tomb elemei
1. elem: 10                   Index   Elem
2. elem: 6                      0        3
3. elem: 2                      1        8
4. elem: 8                      2        4
                                3        0
                                4        6
```

Írjunk programot, amely beolvas egy elemszámot a tömb dinamikus létrehozásához! Keressük ki a tömb legnagyobb elemét és majd ezzel az értékkel osszuk le a tömb minden elemét! Írassuk ki a módosított tömböt! (*Tomb_04*)

```
#include <iostream>
using namespace std;
int main()
{
  double *pw, max;
  int db;
  cout << "Elemek szama: "; cin >> db;
  pw = new double[db];
  for (int i = 0; i<db; i++)
  {
    cout << i << ". elem: "; cin >> pw[i];
  }
  max = pw[0]; // a legnagyobb adatkeresés kezdőértékének beállítása
  for (int i = 1; i<db; i++)
  {
    if (pw[i] > max) max = pw[i]; // átírás, ha van nagyobb
  }
  cout << endl << "Legnagyobb adat: " << max << endl;
  cout << "Legnagyobb adattal normalt adatok:" << endl;
  for (int i =0; i<db; i++)
  {
     pw[i] /= max;   // normálás a legnagyobb adattal
     cout << pw[i] << endl;
  }
  delete [] pw;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 5                 Legnagyobb adat: 10
0. elem: 5                      Legnagyobb adattal normalt adatok:
1. elem: 10                     0.5
2. elem: 6                      1
3. elem: 2                      0.6
4. elem: 8                      0.2
                                0.8
```

5.1.2 Többdimenziós tömbök

A C++ nyelvben is használhatunk többdimenziós tömböket, melyek általános definíciója:

típus tömbnév[méret1][méret2]...[méretn];

Írjunk programot, amely egy 3x3-as egész számokat tartalmazó mátrixot feltölt sorfolyamatosan 1-9-ig. Szorozzuk össze a főátlóbeli elemeket! (*Tomb_05*)

Megjegyzés:

- Egy szimmetrikus mátrix főátlóbeli elemének a sor- és az oszlopindexe megegyezik:
  `i == j`
- A mátrix sorait az *i*, oszlopait a *j* ciklus változtatásával érjük el a programban: `x[i][j]`

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  int m[3][3], k=1, szorzat = 1;
  cout << "Matrix elemeinek feltoltese" << endl << endl;
  for (int i = 0; i<3; i++)
  {
   for (int j = 0; j<3; j++)
   {
     m[i][j] = k + j;
     cout << setw(4) << m[i][j];
   }
    k += 3; cout << endl;
  }
  for (int i = 0; i<3; i++)
   for (int j = 0; j<3; j++)
   {
      if (i == j) szorzat *= m[i][j];
   }
   cout << endl << "Foatlobeli elemek szorzata: "
        << szorzat << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Matrix elemeinek feltoltese

  1   2   3
  4   5   6
  7   8   9

Foatlobeli elemek szorzata: 45
```

5.1.3 Karaktersorozatok (sztringek)

A C++ programban a szövegek tárolására statikus vagy dinamikus helyfoglalású karaktertömböket használunk.

Írjunk programot, amelyben a ***getline()*** tagfüggvénnyel beolvasunk egy mondatot, amely legfeljebb 100 karaktert tartalmazhat! A szavak végződése: *. ! ?* (szóköz, pont, felkiáltójel, kérdőjel) lehet! Számláljuk meg a mondat szavainak számát! A karakterre mutató pointer növelésével írassuk ki a szavakat! (*Tomb_06*)

Megjegyzés:

- A **char** tömb azonosítója a címet tartalmazza, ezért a karakterre mutató pointert a következő utasítással állítjuk rá:

  `pmondat = mondat;`

- A **for** ciklust a feltétel hamis értéke állítja le, ez akkor következik be, amikor a feltételében szereplő *mondat* karaktertömb *i*-edik eleme a szöveg végét jelző nulla értékű ('\0') karaktert tartalmazza:

  `for (int i = 0; mondat[i]; i++)`

  Nem szükséges a `mondat[i]!=0` feltétel használata.

- A mutató léptetése a cím növelését jelenti, azaz a következő karakterre lépünk:

  `pmondat++;` vagy `++pmondat;`

```
#include <iostream>
using namespace std;
int main()
{
  char mondat[100], *pmondat;
  int szavakszama = 0;
  cout << "Mondat szovege: ";
  cin.getline(mondat,100);
  for (int i = 0; mondat[i]; i++)
  {
     switch (mondat[i])
     {
       case ' ': szavakszama++; break;
       case '.': szavakszama++; break;
       case '!': szavakszama++; break;
       case '?': szavakszama++; break;
     }
  }
  cout << endl << "Szavak szama: " << szavakszama << endl;
  cout << endl << "Mondat szavai: " << endl << endl;
  pmondat = mondat;
  while (*pmondat)
  {
    switch (*pmondat)
    {
      case ' ': cout << endl; break;
      case '.': cout << endl; break;
      case '!': cout << endl; break;
      case '?': cout << endl; break;
      default: cout << *pmondat;
   }
   pmondat++;
  }
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Mondat szovege: Ma szep az ido.

Szavak szama: 4

Mondat szavai:

Ma
szep
az
ido
```

5.2 Felhasználó általdefiniált adattípusok

A C++ megengedi, hogy a nyelv meglévő típusait felhasználva újabb típusokat hozzunk létre.

5.2.1 A struct struktúratípus

A C++ nyelven a struktúra (**struct**) típus több tetszőleges típusú (kivéve a **void** és a függvénytípust) adatok együttese. Általános formája:

struct *struktúratípus* {
típus1 tag1; *// kezdőérték nélkül*
típus2 tag2;
. . .
típusN tagN;
};

Írjunk programot, amely beolvassa a *teglalap* típusú struktúra *a* és *b* tagját statikus és dinamikus helyfoglalású változók esetében! Számítsuk ki a téglalap kerületét és területét! (*Struct_01*)

Megjegyzés:

- A **struct** kulcsszó melletti név a típus, azzal definiálunk: `teglalap t1;`
- A struktúra befejező } zárójele mögötti név, már definíció, azaz a példában a *t2*-nek már van helye a memóriában: `t2.a = 8;`
- Statikus struktúra esetében az adattagokat ponttal érjük el:

```
teglalap t1;
t1.a = 2;
```

- Dinamikus struktúra esetén, először helyet kell foglalni a struktúra számára a memóriában a **new** oprátorral. A tagokat `->` (kötőjel és nagyobb jel) érjük el. A dinamikus helyfoglalású struktúrát a **delete** operátorral töröljük:

```
pt2 = new teglalap;
pt2->a = 4; pt2->b = 5;
delete pt2;
```

```
#include <iostream>
using namespace std;
struct teglalap{
                  double a, b;
                  double ter, ker;
                }t2;
int main()
{
  teglalap t1, *pt2;
  t1.a = 2; t1.b = 3;
  t1.ter = t1.a * t1.b;
  t1.ker = 2*(t1.a + t1.b);
  cout << "1. Teglalap" << endl;
  cout << "teglalap oldalai: " << t1.a << "  " << t1.b << endl;
  cout << "kerulet: " << t1.ker << endl;
  cout << "terulet: " << t1.ter << endl;
  t2.a = 8; t2.b = 7;
  t2.ter = t2.a * t2.b;
  t2.ker = 2*(t2.a + t2.b);
  cout << endl << "2. Teglalap" << endl;
  cout << "teglalap oldalai: " << t2.a << "  " << t2.b << endl;
  cout << "kerulet: " << t2.ker << endl;
  cout << "terulet: " << t2.ter << endl;
  pt2 = new teglalap;
  pt2->a = 4; pt2->b = 5;
  pt2->ter = pt2->a*pt2->b;
  pt2->ker = 2*(pt2->a+pt2->b);
  cout << endl << "3. Teglalap" << endl;
  cout << "teglalap oldalai: " << pt2->a << "  " << pt2->b << endl;
  cout << "kerulet: " << (*pt2).ker << endl;
  cout << "terulet: " << (*pt2).ter << endl;
  delete pt2;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. Teglalap
teglalap oldalai: 2  3
kerulet: 10
terulet: 6

2. Teglalap
teglalap oldalai: 8  7
kerulet: 30
terulet: 56

3. Teglalap
teglalap oldalai: 4  5
kerulet: 18
terulet: 20
```

5.2.2 Struktúratömbök

Struktúratömböt pontosan úgy kell definiálni, mint bármilyen más tömböt. Definiálhatjuk statikusan, vagy dinamikusan. A tömb elemei struktúrák.

Írjunk programot, amely beolvassa a definícióban megadott *M* elemű, *teglalap* típusú *t* tömb *a* és *b* adattagjait! Számítsuk ki a téglalapok kerületét és területét! Keressünk meg a legkisebb kerületet és a legnagyobb területet! (*Struct_02*)

Megjegyzés:
A példában szereplő *t* tömb eleme *teglalap* típusú struktúra, akkor a struktúra tagjait a következő módon érhetjük el:

```
t[i].a = 2;
```

azaz a tömbből az indexeléssel kiválasztjuk a struktúrát, majd pedig a pont operátorral kijelölhetjük a struktúra tagját.

```
include <iostream>
using namespace std;
struct teglalap{
                 double a, b;
                 double ter, ker;
               };
const int M = 5;
int main()
{
 teglalap t[M];
 double maxter, minker;
 cout << "Teglalap adatainak beolvasasa" << endl;
 for (int i = 0; i<M; i++)
 {
   cout << i << ".elem a oldala : "; cin >> t[i].a;
   cout << i << ".elem b oldala : "; cin >> t[i].b;
   cout << endl;
```

```
    t[i].ter = t[i].a * t[i].b;
    t[i].ker = 2*(t[i].a + t[i].b);
  }
  maxter = t[0].ter;
  minker = t[0].ker;

  for (int i = 0; i<M; i++)
  {
    if (t[i].ter > maxter) maxter = t[i].ter;
    if (t[i].ker < minker) minker = t[i].ker;
  }
  cout << endl;
  cout << "Min. kerulet: " << minker << endl;
  cout << "Max. terulet: " << maxter << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Teglalap adatainak beolvasasa
0.elem a oldala : 1
0.elem b oldala : 1

1.elem a oldala : 2
1.elem b oldala : 3

2.elem a oldala : 4
2.elem b oldala : 5

3.elem a oldala : 3
3.elem b oldala : 2

4.elem a oldala : 2
4.elem b oldala : 6

Min. kerulet: 4
Max. terulet: 20
```

Írjunk programot, amely beolvassa az adatok darabszámát és helyet foglal a definícióban megadott `teglalap *t` tömbnek! Olvassuk be a téglalapok *a* és *b* oldalait. Számítsuk ki a téglalapok területét és kerületét, keressük ki a legkisebb kerületet és a legnagyobb területet! Szabadítsuk fel a tömb számára lefoglalt memóriaterületet! (*Struct_03*)

Megjegyzés:

- A példában a `teglalap *t;` a *t* téglalap struktúrára mutató pointer. Ha *db* darab struktúrának foglalunk helyet, akkor a következő műveletet kell végrehajtanunk a **new** operátorral:

```
t = new teglalap[db];
```

A **new** után a típust, amely most a *teglalap*, és a szögletes zárójelben a darabszámot [*db*] kell megadni.

- A tömb törlése a memóriából:

  ```
  delete [] t;
  ```

```
#include <iostream>
using namespace std;
struct teglalap{
                  double a, b;
                  double ter, ker;
                 };
int main()
{
 teglalap *t;
 int db;
 double maxter, minker;
 cout << "Elemek szama: "; cin >> db;
 t = new teglalap[db];
 cout << "Teglalap adatainak beolvasasa" << endl;
 for (int i = 0; i<db; i++)
 {
   cout << i << ".elem a oldala : "; cin >> t[i].a;
   cout << i << ".elem b oldala : "; cin >> t[i].b;
   cout << endl;
   t[i].ter = t[i].a * t[i].b;
   t[i].ker = 2*(t[i].a + t[i].b);
 }
 maxter = t[0].ter;
 minker = t[0].ker;
 for (int i = 1; i<db; i++)
 {
   if (t[i].ter > maxter) maxter = t[i].ter;
   if (t[i].ker < minker) minker = t[i].ker;
 }
 cout << endl;
 cout << "Min. kerulet: " << minker << endl;
 cout << "Max. terulet: " << maxter << endl;
 delete [] t;
 cin.get(); cin.get();
 return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 3                     Min. kerulet: 6
Teglalap adatainak beolvasasa       Max. terulet: 30
0.elem a oldala : 2
0.elem b oldala : 3

1.elem a oldala : 1
1.elem b oldala : 2

2.elem a oldala : 5
2.elem b oldala : 6
```

Írjunk programot, amely konstanssal feltöltött *teglalap* típusú struktúratömb adatait jeleníti meg! Számítsuk ki a téglalapok területét és kerületét, keressük meg a legnagyobb területet és a legkisebb kerületet! (*Struct_04*)

```
#include <iostream>
using namespace std;
struct teglalap{
                  double a, b;
                  double ter, ker;
                };
 teglalap t[] ={ {2, 3},
                 {5, 4},
                 {6, 2},
                 {7, 5} };
const int db = sizeof(t)/sizeof(t[0]);
int main()
{
 double maxter, minker;
 cout << "Teglalap adatainak megjelenitese" << endl;
 for (int i = 0; i<db; i++)
 {
   cout << endl << "a oldal: " << t[i].a << endl;
   cout << "b oldal: " << t[i].b << endl;
   t[i].ter = t[i].a * t[i].b;
   t[i].ker = 2*(t[i].a + t[i].b);
 }
 maxter = t[0].ter;
 minker = t[0].ker;
 for (int i = 0; i<db; i++)
 {
   if (t[i].ter > maxter) maxter = t[i].ter;
   if (t[i].ker < minker) minker = t[i].ker;
 }
 cout << endl;
 cout << "Min. kerulet: " << minker << endl;
 cout << "Max. terulet: " << maxter << endl;
 cin.get();
 return 0;
}
```

A program futásának eredménye:

```
Teglalap adatainak megjelenitese    Min. kerulet: 10
                                    Max. terulet: 35
a oldal: 2
b oldal: 3

a oldal: 5
b oldal: 4

a oldal: 6
b oldal: 2

a oldal: 7
b oldal: 5
```

Írjunk programot, amely beolvassa a *teglalap_terker* típusú *teglalap1* változó *t* tagjának adatait! A *t* tag szintén egy struktúra. Számítsuk ki a téglalap területét és kerületét! (*Struct_05*)

Megjegyzés:

- Ha egy struktúra adattagja szintén egy struktúra, akkor az adatok elérése a következő módon történik:

  ```
  teglalap_terker teglalap1;
  teglalap1.t.a = 2;
  ```

  Azaz első pont után a tagnév, és a második pont után a beágyazott struktúra tagneve következik.

```
#include <iostream>
using namespace std;
struct teglalap_adatok {
                         double a, b;
                        };
struct teglalap_terker {
                          teglalap_adatok t;
                          double ter, ker;
                        };
int main()
{
  teglalap_terker teglalap1;
  teglalap1.t.a = 2; teglalap1.t.b = 3;
  teglalap1.ter = teglalap1.t.a * teglalap1.t.b;
  teglalap1.ker = 2*(teglalap1.t.a + teglalap1.t.b);
  cout << "Teglalap adatai" << endl;
  cout << "a oldal: " << teglalap1.t.a
       << "\nb oldal: " << teglalap1.t.b << endl;
  cout << "\nkerulet: " << teglalap1.ker << endl;
  cout << "terulet: " << teglalap1.ter << endl;

  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Teglalap adatai
a oldal: 2
b oldal: 3

kerulet: 10
terulet: 6
```

Írjunk programot, amely egymásba ágyazott struktúrák használatával elvégzi a négy alapműveletet! (*Struct_06*)

```
#include <iostream>
#include <cstring>
using namespace std;
struct Adatok{
               double a,b;  // operandusok
             };
struct MuveletiJel{
                    char muv;  // műveleti jel
                  };
struct Muvelet {
                 Adatok op;
                 MuveletiJel muvjel;
                 double eredmeny;
               };
int main()
{
  Muvelet m;
  cout << "1. adat: ";  cin >> m.op.a;
  cout << "2. adat: ";  cin >> m.op.b;
  do
  {
    cout << "muveleti jel (+,-,*,/): "; cin >> m.muvjel.muv;
  }while (! strchr("+-*/",m.muvjel.muv));
  switch (m.muvjel.muv)
  {
   case '+' : m.eredmeny = m.op.a + m.op.b; break;
   case '-' : m.eredmeny = m.op.a - m.op.b; break;
   case '*' : m.eredmeny = m.op.a * m.op.b; break;
   case '/' : m.eredmeny = m.op.a / m.op.b; break;
  }
  cout << m.op.a << " " << m.muvjel.muv << " "
       << m.op.b << " = " << m.eredmeny << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. adat: 3.5
2. adat: 2
muveleti jel (+,-,*,/): *
3.5 * 2 = 7
```

6. Függvények

A C++ program általában függvények sorozatából épül fel, amely tartalmazza a nélkülözhetetlen ***main()*** függvényt.

6.1 Függvények definíciója és deklarációja

A saját készítésű függvényeinket mindig definiálni kell. A definíció, amelyet csak egyszer lehet megadni, a C++ programon belül bárhol elhelyezkedhet.

A függvény-definíció általános formája:

```
<visszatérési típus> függvénynév(<paraméter-deklarációs lista>)
{
 // a függvény törzse
 <lokális definíciók és deklarációk>
 <utasítások>
}
```

A függvény deklarációja (prototípusa) megelőzi a függvény definícióját. A prototípus a függvény fejléce, pontosvesszővel lezárva:

```
<visszatérési típus> függvénynév(<paraméter-deklarációs lista>);
```

Függvény prototípusában a paramétereknek nincs jelentőségük, ezért az alábbi két prototípus azonos:

```
double Szorzat1(double x, double y);
double Szorzat1(double, double);
```

6.2 A függvények paraméterezése és a függvényérték

A paraméterlista elemeit gyakran formális paramétereknek nevezzük, amelyek akkor válnak aktuálissá, amikor hívjuk a függvényt. A C++ nyelv elnevezése alapján a függvény definíciójában paramétereket használunk, míg a hívásában argumentumokkal dolgozunk.

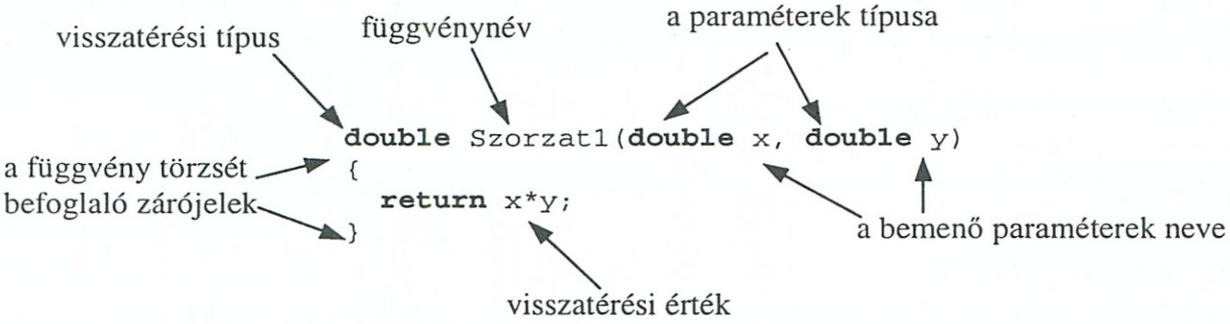
Működés szempontjából a függvényeknek kétféle paramétere lehet:

- Bemenő paraméterek, melyek értéket továbbítanak a függvény felé.
- Kimenő paraméterek, melyek a függvény törzsében kapott értéket közvetítik a függvény hívójának. (Logikailag ide tartozik a függvényérték is.)
  - A függvény a **return** mellett csak egy értéket képes visszaadni, melynek típusát a függvény neve előtti típus határozza meg.

- Ha a függvény nem a **return** mellett adja vissza az eredményt, akkor a függvény neve előtt a **void** általános típus szerepel. A paraméterlistán csak pointer (melyet a C++ nyelvben kevésbé használjuk), vagy a C++ nyelvben sokkal kényelmesebben alkalmazható, a referencia (hivatkozás) segítségével adhatjuk vissza az eredményeket.

A mintafeladatban két függvényen vizsgáljuk meg a függvény jellemzőit:

1. A *Szorzat1()* függvény alapján:



```
double Szorzat1(double x, double y)
{
   return x*y;
}
```

visszatérési típus
függvénynév
a paraméterek típusa
a függvény törzsét befoglaló zárójelek
a bemenő paraméterek neve
visszatérési érték

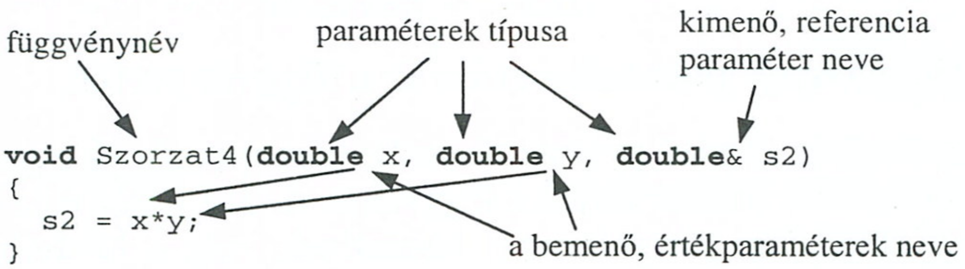
- Általában a bemenő paraméterek értékével számolunk a függvény törzsében, és különféle eredményeket hozhatunk létre.
- A függvény a nevén keresztül (a **return** utasítással) egy értéket tud visszaadni, melynek típusát a függvény neve előtt kell meghatározni, például:

  double *Szorzat1*(**double**, **double**);

  A függvény belsejében pedig a **return** x*y; utasítás szolgáltatja a visszatérési értéket. A függvény hívása ebben az esetben:

  ```
  ered1 = Szorzat1(a,b);
  ```

2. A *Szorzat4()* függvény alapján:



```
void Szorzat4(double x, double y, double& s2)
{
   s2 = x*y;
}
```

függvénynév
paraméterek típusa
kimenő, referencia paraméter neve
a bemenő, értékparaméterek neve

- A **void** típusú függvényekhez is használhatjuk a **return** utasítást, azonban kifejezés nélkül: **return**; lehet. Az ilyen függvénynek hívása önállóan szerepel az utasításban, például:

  ```
  Szorzat4(a, b, ered3);
  ```

- A függvénynek lehet több kimenő paramétere is. Ezeket pointer- vagy referenciaparaméter segítségével lehet visszaadni.

6.3 A függvényhívás

Foglaljuk össze a függvényhívás szabályait!

1. Ha a függvény neve előtt **void**-tól eltérő visszatérési típus szerepel, akkor ennek a típusnak megfelelő típusú változó fogadhatja a függvény **return** utasítással visszaadott értékét.
   Ha a prototípus:

   ```
   double Szorzat1(double x, double y);
   ```

   akkor a függvény által visszaadott értéket az értékadó operátor bal oldalán fogadhatjuk:

   ```
   double a =2, b = 4, ered1, ered3;
   ered1 = Szorzat1(a, b);
   ```

2. Ha a prototípus **void**:

   ```
   void Szorzat4(double x, double y, double& s2);
   ```

   Az eredmény fogadása a harmadik paraméter feladata. A függvényt az *ered3* változóval aktiváljuk, azaz átadjuk a hivatkozását, hogy a függvény bele tudjon írni:

   ```
   double a =2, b = 4, ered1, ered3;
   Szorzat4(a, b, ered3);
   ```

Írjunk függvényeket, amelyek többféle módon számítják ki két valós szám szorzatát, valamint egy függvényt, amely két valós adatot olvas be! Írjunk főfüggvényt a függvények aktiválására! (*Minta_fg*)

A mintaprogramban szereplő függvények prototípusai és feladatai:

Írjunk *Szorzat1()* függvényt, amely a paraméterlistán megadott két valós szám szorzatát először segédváltozóban tárolja, majd visszatérési értékként, a **return** mellett adja vissza!

```
double Szorzat1(double x, double y);
```

Írjunk *Szorzat2()* függvényt, amely a paraméterlistán megadott két valós szám szorzatát visszatérési értékként, közvetlenül a **return** mellett adja vissza!

```
double Szorzat2(double x, double y);
```

Írjunk *Szorzat3()* függvényt, amely a paraméterlistán szereplő két valós szám szorzatát s1 mutató segítségével adja vissza!

```
void Szorzat3(double x, double y, double *s1);
```

Írjunk *Szorzat4()* függvényt, amely a paraméterlistán megadott két valós szám szorzatát referenciaként megadott *s2* paraméterben adja vissza:

```
void Szorzat4(double x, double y, double& s2);
```

Írjunk *Olvas()* függvényt, amely a két beolvasott valós számot referencia-paraméterben adja vissza!

```
void Olvas(double& x, double& y);
```

Írjunk főprogramot, amely a függvényeket aktiválja, és jelenítsük meg az eredményeket. Használjuk az alábbi definíciót!

```
double a, b, ered1, ered2, ered3, ered4;
```

A feladat megoldása

```
//MINTA_FG
#include <iostream>
using namespace std;
double Szorzat1(double x, double y);
double Szorzat2(double x, double y);
void Szorzat3(double x, double y, double *s1);
void Szorzat4(double x, double y, double& s2);
void Olvas(double& x, double& y);
int main()
{
  double a, b, ered1, ered2, ered3, ered4;
  cout << "1. adat: "; cin >> a;
  cout << "2. adat: "; cin >> b;
  ered1 = Szorzat1(a,b);
  cout << "Szorzat1 eredmenye: " << ered1 << endl;
  Szorzat3(a, b, &ered2); // az ered2 változó címét kell átadni
  cout << "Szorzat3 eredmenye: " << ered2 << endl;
  Szorzat4(a, b, ered3);
  cout << "Szorzat4 eredmenye: " << ered3 << endl;
  Olvas(a,b);
  ered4 = Szorzat2(a,b);
  cout << "Szorzat2 eredmenye(uj adattal): " << ered4 << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. adat: 2
2. adat: 3
Szorzat1 eredmenye: 6
Szorzat3 eredmenye: 6
Szorzat4 eredmenye: 6
1. adat: 4
2. adat: 5
Szorzat2 eredmenye(uj adattal): 20
```

A függvények megoldása magyarázattal:

A paraméterlistán átadott két szám szorzatát négyféleképpen adjuk vissza.

Mivel a függvény visszatérési értékének a típusa **double**, ezért lokálisan definiálunk egy **double** típusú *s* változót. Az *x* és az *y* paraméter szorzatát az *s* változóba számítjuk ki, melyet azután a **return** mellett adunk vissza:

```
double Szorzat1(double x, double y)
{
   double s;
   s = x * y;
   return s;
}
```

Hívása: `ered1 = Szorzat1(a,b);`

Mivel nagyon rövid és egyszerű a kifejezés, amely az eredményt szolgáltatja, ezért a **return** mellett közvetlenül is megadhatjuk:

```
double Szorzat2(double x, double y)
{
  return x * y;
}
```

Hívása: `ered4 = Szorzat2(a,b);`

A *Szorzat3()* függvény a paraméterlistán megadott két szám szorzatát számítja ki. Az eredményt a paraméterlistán egy mutató segítségével adjuk vissza. Ahová az *s1* mutat, oda számítjuk ki az *x* és az *y* paraméterben megadott adatok szorzatát.

```
void Szorzat3(double x, double y, double *s1)
{
  *s1 = x * y;
}
```

Hívása: `Szorzat3(a, b, &ered2);`

A harmadik paraméter **double*** mutató pointer, ezt az *&ered2*, azaz az *ered2* címével helyettesítjük.

A *Szorzat4()* függvény szintén a megadott két szám szorzatát számítja ki. Az eredményt a paraméterlistán, a referenciaként megadott *s2* változóban adjuk vissza. A függvény hívásakor az *s2* lesz a szinonim neve az argumentumként megadott **double** típusú változónak.

```
void Szorzat4(double x, double y, double& s2)
{
  s2 = x * y;
}
```

Hívása: `Szorzat4(a, b, ered3);`

A referenciával megadott paramétert azonos típusú argumentummal kell helyettesíteni.

Az adatok beolvasására is írhatunk függvényt. Az *Olvas* függvény rákérdez szövegesen az adatokra, melyeket a paraméterlistán referenciaként megadott változókban ad vissza.

```
void Olvas(double& x, double& y)
{
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
}
```

6.4 Különböző típusú paraméterek

A függvények a legkülönfélébb típusú paramétereket tudnak kezelni. E fejezet feladatai bemutatják a használatukat.

6.4.1 Aritmetikai típusú paraméterek

Írjunk programot, amely függvényekkel oldja meg egy derékszögű háromszög két befogójának beolvasását, átfogójának, területének és kerületének számítását! (*Atfogo*)

```
#include <iostream>
#include <cmath>
using namespace std;
void AdatotOlvas(double& a, double& b);
void AtfogoSzamitasa(double a, double b, double& c);
double TerKer(double a, double b, double c, double& ker);
int main()
{
  double befogo1, befogo2, atfogo, kerulet, terulet;
  AdatotOlvas(befogo1,befogo2);
  AtfogoSzamitasa(befogo1,befogo2, atfogo);
  terulet = TerKer(befogo1, befogo2, atfogo,kerulet);
  cout << endl << "Atfogo: " << atfogo << endl;
  cout << endl << "Terulet: " << terulet << endl;
  cout << "Kerulet: " << kerulet << endl;
  cin.get(); cin.get();
  return 0;
}
```

Az *AdatotOlvas()* függvény referenciaként megadott két paraméterében adja vissza a derékszögű háromszög két befogójának (*a,b*) értékét, melyet a billentyűzetről olvas be.

```
void AdatotOlvas(double& a, double& b)
{
  cout << "Derekszogu haromszog adatai" << endl;
  cout << "Befogo1: "; cin >> a;
  cout << "Befogo2: "; cin >> b;
}
```

Az *AtfogoSzamitasa()* függvény bejövő paraméterei a derékszögű háromszög két befogója: *a,b*. Kimenő paramétere a referenciaként megadott *c*, amelybe a két befogó négyzetösszegéből vont négyzetgyökkel kiszámítja az átfogót.

```
void AtfogoSzamitasa(double a, double b, double& c)
{
  c = sqrt(pow(a,2) + pow(b,2));
}
```

A *TerKer()* függvény a derékszögű háromszög három oldala alapján a referenciaként megadott (kimenő paraméter) *ker* változóba számítja ki a háromszög kerületét. A területet pedig visszatérési értékként adja vissza.

```
double TerKer(double a, double b, double c, double& ker)
{
  ker = a + b + c;
  return (a * b)/2;
}
```

A program futásának eredménye:

```
Derekszogu haromszog adatai
Befogo1: 3
Befogo2: 4

Atfogo: 5

Terulet: 6
Kerulet: 12
```

6.4.2 Felhasználói típusú paraméterek

Írjunk programot, amely függvénnyel olvassa be a téglalap oldalainak hosszát és kiszámítja a téglalap területét és kerületét! Az adatok tárolására használjunk struktúrát! (*Teglalap*)

```
#include <iostream>
using namespace std;
struct teglalap {                   // a teglalap struktúra definíciója
                  double a,b;       // a téglalap oldalai
                  double ter,ker;   // a téglalap területe és kerülete
                };
void Olvas(teglalap& t);
void Szamol(teglalap& t);
int main()
{ teglalap tg;
  Olvas(tg);
  Szamol(tg);
  cout << endl << "Teglalap kerulete: " << tg.ker << endl;
  cout << "Teglalap terulete: " << tg.ter << endl;
  cin.get(); cin.get();
  return 0;
}
```

Az *Olvas()* függvény egy *teglalap* típusú struktúrát vár argumentumként. Rákérdez a téglalap oldalaira és beolvassa azokat.

```
void Olvas(teglalap& t)
{ cout << "Teglalap adatai " << endl << endl;
  cout << "a oldal: "; cin >> t.a;
  cout << "b oldal: "; cin >> t.b;
}
```

A *Szamol()* függvény szintén egy *teglalap* típusú struktúrát fogad paraméterként. Kiszámítja a téglalap területét szintén egy struktúra *ter* tagjába, kerületét pedig a struktúra *ker* tagjába.

```
void Szamol(teglalap& t)
{
  t.ter = t.a * t.b;
  t.ker = 2*(t.a + t.b);
}
```

A program futásának eredménye:

```
Teglalap adatai

a oldal: 3
b oldal: 4

Teglalap kerulete: 14
Teglalap terulete: 12
```

6.4.3 Tömbök átadása függvénynek

Írjunk programot, amelyben a *TombOlvas()* függvénnyel olvassuk be az egész típusú tömb adatait, és a *Keres()* függvénnyel megszámláljuk, hogy egy megadott adat a tömb elemei között hányszor fordult elő! (*AdatKeres*)

```
#include <iostream>
using namespace std;
void TombOlvas(int t[], int& n);
void Keres(const int t[], int n, int adat, int& talalat);
int main()
{
  int v[20], db, keres, talalt;
  TombOlvas(v,db);
  cout << endl << "Keresendo adat: "; cin >> keres;
  Keres(v,db,keres, talalt);
  cout << "Megtalalt adatok szama: " << talalt << endl;
  cin.get(); cin.get();
  return 0;
}
```

A *TombOlvas()* függvény a paraméterlistán adja vissza az adatokkal feltöltött egész típusú *t* tömböt és a tömb aktuális elemeinek darabszámát (*n*). A függvény hibás darabszámot nem fogad el.

```
void TombOlvas(int t[], int& n)
{
  do
  { cout <<"Elemek szama (max.20): "; cin >> n;
  }while (0>= n || n > 20);
  for (int i = 0; i< n; i++)
  {
    cout << i << ". elem: ";
    cin >> t[i];
  }
}
```

A *Keres()* függvény a *t* tömb és az *n* darabszám ismeretében a megadott *adat*ra vizsgálatot végez, hogy hányszor fordul elő a tömb elemei között, és azt a *talalat* kimenő paraméterben adja vissza. A paraméterlistán azért szerepel a tömb **const** előírással, azaz konstansként megadva, hogy a tömb elemeit ne lehessen megváltoztatni a függvényen belül.

```
void Keres(const int t[], int n, int adat, int& talalat)
{
  talalat = 0;
  for (int i = 0; i<n; i++)
  {
    if (t[i] == adat) talalat++;
  }
}
```

A program futásának eredménye:

```
Elemek szama (max.20): 5
0. elem: 2
1. elem: 4
2. elem: 3
3. elem: 1
4. elem: 7

Keresendo adat: 3
Megtalalt adatok szama: 1
```

Írjunk programot, amelyben a *TombOlvas()* függvénnyel olvassuk be a saját típussal definiált valós tömböt, és a *KorlatKeres()* függvénnyel számláljuk meg, hogy a tömbnek hány eleme esik egy adott korlát fölé! (*Korlat*)

```
#include <iostream>
using namespace std;
typedef double tombtp[20];
void TombOlvas(tombtp t, int db);
int KorlatKeres(tombtp t, int db, double korlat);
int main()
{
  tombtp x;
  double x_korlat;
  int x_db, maxelem, x_korlat_db;
  maxelem = sizeof(tombtp)/sizeof(x[0]); // tömb méretének kiszámítása
                                          // maxelem értéke 20
  do
  {
    cout << "Elemek szama: "; cin >> x_db;
  } while (0 >= x_db || x_db > maxelem);
  TombOlvas(x,x_db);
  cout << endl << "Korlat: "; cin >> x_korlat;
  x_korlat_db = KorlatKeres(x, x_db, x_korlat);
  cout << "Korlat feletti elemek szama: " << x_korlat_db;
  cin.get(); cin.get();
  return 0;
}
```

A *TombOlvas()* függvény tömb aktuális elemeinek *db* darabszámát a függvény paraméterlistán kapja, és a paraméterlistán adja vissza az adatokkal feltöltött *tombtp* típusú *t* tömböt.

```
void TombOlvas(tombtp t,int db)
{
  for (int i = 0; i<db; i++)
  {
    cout << i << ".elem: "; cin >> t[i];
  }
}
```

A *KorlatKeres()* függvény a *db* elemszámú *tombtp* típusú tömbnek a *korlat* paraméternél nagyobb elemeit megszámlálja a *szamlal* lokális változóban, és a kapott darabszámot a **return** mellett, függvényértékként adja vissza.

```
int KorlatKeres(tombtp t, int db, double korlat)
{
   int szamlal = 0;
   for (int i = 0; i<db; i++)
   {
     if (t[i] > korlat) szamlal++;
   }
   return szamlal;
}
```

A program futásának eredménye:

```
Elemek szama: 4
0.elem: 2.5
1.elem: 3.8
2.elem: 4.2
3.elem: 5.6

Korlat: 3.9
Korlat feletti elemek szama: 2
```

Írjunk programot, amely függvények hívásával dinamikus tömböt hoz létre. Töltsük fel adatokkal a tömböt, a tömb elemeit normáljuk egy megadott adattal, valamint listázzuk ki a tömb elemeit! (*Normal*)

```
#include <iostream>
using namespace std;
double* TOlvas(int& db);
double* TNormal(double *w, int db, double normal);
void TKiir(const double *w, int db);
int main()
{
  double *y, y_normal;
  int y_db;
  y = TOlvas(y_db);
  cout << endl << "Normalo adat: "; cin >> y_normal;
  y = TNormal(y,y_db,y_normal);
  TKiir(y, y_db);
  delete [] y;
  cin.get(); cin.get();
  return 0;
}
```

A *TOlvas()* függvény a paraméterben adja vissza a tömb elemeinek darabszámát, a dinamikusan létrehozott és adatokkal feltöltött valós tömb mutatóját pedig visszatérési értékként szolgáltatja.

```
double* TOlvas(int& db)
{
   double *w;
   cout << "Elemek szama: "; cin >> db;
   w = new double[db];
   for (int i = 0; i<db; i++)
   {
     cout << i << ". elem: "; cin >> w[i];
   }
   return w;
}
```

A *TNormal()* függvény a paraméterlistán kapja a valós tömb mutatóját (**w*) és a tömb aktuális darabszámát (*db*). A függvény a tömb minden elemét a *normal* paraméterben megadott adattal leosztja, az így megváltozott tömb mutatóját függvényértékként adja vissza.

```
double* TNormal(double *w, int db, double normal)
{
   for (int i = 0; i<db; i++)
   {
      w[i] /= normal;
   }
   return w;
}
```

A *Kiir()* függvény a tömb adatait listázza a konstansként megadott tömbre mutató pointer és a tömb adatainak darabszáma ismeretében.

```
void TKiir(const double *w, int db)
{
   cout << endl << "Normalt adatok" << endl;
   for (int i = 0; i<db; i++)
   {
      cout << w[i] << endl;
   }
}
```

A program futásának eredménye:

```
Elemek szama: 5
0. elem: 1.2
1. elem: 2.8
2. elem: 3.2
3. elem: 4.6
4. elem: 5.2

Normalo adat: 1.2

Normalt adatok
1
2.33333
2.66667
3.83333
4.33333
```

Írjunk programot, amely függvények használatával lottósorsolást szimulál és a számokat növekvő sorrendben jeleníti meg! (*Lotto*)

```
#include <iostream>
#include <cstdlib>
#include <ctime>
void Kezdoertek();
void LottoSorsolas(int t[]);
void Rendez(int t[]);
int main()
{ int lotto[5];
  Kezdoertek();
  LottoSorsolas(lotto);
  Rendez(lotto);
  cin.get();
  return 0;
}
```

A *Kezdoertek()* függvény a ***srand*** könyvtári függvénnyel a véletlenszám-generátornak új kezdőértéket állít be.

```
void Kezdoertek()
{ srand((unsigned) time(NULL));}
```

A *LottoSorsolas()* függvény a kihúzás sorrendjében egy *t* egész típusú tömbben adja vissza az 5 darab, 1-90 közötti különböző lottószámot.

```
void LottoSorsolas(int t[])
{
  int  van[91], db = 0, sorsol;
  for (int i = 0; i<91; i++)
  {
    van[i] =0;  // lottószámok nyilvántartásának nullázása
  }

  while (db < 5)
  {
    sorsol = rand()% 90 +1; // lottószám generálása
    if (van[sorsol] == 0)   // lottószámok kihúzásának ellenőrzése
    {
       van[sorsol] = 1;     // ha a számot még nem húzták ki
       t[db++] = sorsol;    // bekerül a t tömb db-edik elemébe
    }
  }
  cout << "Lottosorsolas" << endl;
  for (int i=0; i<5; i++)
  {
    cout << t[i] << endl; // lottószámok kiírása
  }
}
```

A *Rendez()* függvény a paraméterlistán megadott *t* tömb elemeit növekvő sorrendbe rendezve írja ki.

```
void Rendez(int t[])
{
  int seged;
  for (int i = 0; i<4; i++)
    for (int j = i+1; j<5; j++)
    {
      if (t[i] > t[j])
      {
        seged = t[i]; t[i] = t[j]; t[j] = seged;
      }
    }
  cout << endl
       << "Lottosorsolas novekvo sorrendben" << endl;
  for (int i=0; i<5; i++)
  { cout << t[i] << endl; }
}
```

A program futásának eredménye:

```
Lottosorsolas                  Lottosorsolas novekvo sorrendben
72                             12
45                             39
12                             45
39                             68
68                             72
```

6.4.4 Sztringparaméterek

Írjunk programot, amely függvények hívásával egy adott karaktersorozatból egy megadott karaktert töröl, és a törlés helyén összehúzza a karaktereket, valamint a karaktertömböt kiírja a képernyőre! A feladat megoldása során az eredeti és a módosított karaktersorozatra is szükségünk van. (*KarTorol*)

Megjegyzés: C++ nyelvben a karaktersorozatok tárolására karaktertömböt használnak, melyben a karaktereket egy nullás bájt zárja.

```
#include <iostream>
#include <cstring>
using namespace std;
char* KarTorol(char szoveg[], char ch);
void SzovegKiir(char *s);
int main()
{
  char be[80];
  char *ki, c;
  cout << "Szoveg [max. 80 karakter]: "; cin.getline(be,80);
  cout << "Torlendo karakter: "; cin >> c;
  ki = KarTorol(be, c);
  cout << endl << "Modositott szoveg: " << endl;
  SzovegKiir(ki);
  delete ki;
  cin.get(); cin.get();
  return 0;
}
```

A *KarTorol()* függvény a paraméterlistán átadott karaktertömbből kitörli a *ch* karaktert, és a megváltozott tömb címét függvényértékként adja vissza.

```
char* KarTorol(char szoveg[], char ch)
{
  char* kiszoveg;
  int j = 0, db;
  db = strlen(szoveg);          // a karaktersorozat karaktereinek száma
  kiszoveg = new char[db+1];    // a módosított szöveget tartalmazza
                           // db+1 a sztringet záró '\0' miatt szükséges

  for (int i=0; i<db; i++)
  {
    if (szoveg[i] != ch)
    {
      kiszoveg[j++] = szoveg[i]; // átmásolja az i-edik karaktert
    }
  }
  kiszoveg[j] = '\0'; // lezárja a sztringet
  return kiszoveg;
}
```

A *SzovegKiir()* függvény kiírja a paraméterlistán átadott karaktersorozatot.

```
void SzovegKiir(char *s)
{
 for (int i = 0; *s;i++)
   cout << *s++;  // az s címét növeli
 cout << endl;
}
```

A program futásának eredménye:

```
Szoveg [max. 80 karakter]: ma az ido esos szeles
Torlendo karakter: e

Modositott szoveg:
ma az ido sos szls
```

6.4.5 A függvény, mint argumentum

Írjunk programot, amely függvény hívásával a paraméterlistán átadott függvénynek kiszámítja az integrálját az adott alsó és felső határ között a téglalapintegrál módszerével! (*Integral*)

```
#include <iostream>
using namespace std;
typedef double fv(double x);
typedef fv *fp;
double fg1(double x);
double TeglalapIntegral(double a, double b, int n, fp f);
int main()
{
  double alsoH, felsoH, IntegralErteke;
  int osztas;
  cout << "Integral szamitas teglalap modszerrel" << endl << endl;
  cout << "Also hatar: "; cin >> alsoH;
  cout << "Felso hatar: "; cin >> felsoH;
  cout << "Intervallum felosztasa: "; cin >> osztas;
  IntegralErteke = TeglalapIntegral(alsoH, felsoH, osztas, fg1);
  cout << endl << "Integral erteke: " << IntegralErteke << endl;
  cin.get(); cin.get();
  return 0;
}
```

Az *fg1()* függvény számítja ki a bejövő paraméterben megadott *x* helyen az integrálandó függvény helyettesítési értékét, amelyet függvényértékként ad vissza.

```
double fg1(double x)
{
   return 4.0/(1. + x * x);
}
```

A *TeglalapIntegral()* függvény paraméterként megkapja az integrálandó függvény alsó (*a*) és felső (*b*) határát, valamint az osztások számát (*n* páros), és kiszámítja az integrált téglalap módszerrel.

```
.double TeglalapIntegral(double a, double b, int n, fp f)
{
  double s = 0, h = (b-a)/n;
  for (int i = 1; i<n; i++)
    s += (*f)(a+h*i);      // f(a+h*i); függvényhívással azonos
  }
  return h*((*f)(a) + s + (*f)(b));
}
```

A program futásának eredménye:

```
Integral szamitas teglalap modszerrel

Also hatar: 0
Felso hatar: 1
Intervallum felosztasa: 3000

Integral erteke: 3.14259
```

6.4.6 Alapértelmezés szerinti (default) argumentumok

A C++ függvények prototípusában a paraméterekhez ún. alapértelmezés szerinti értékeket rendelhetünk. Az alapértelmezés szerinti értékkel ellátott paraméterek *jobbról-balra* haladva, folyamatosan helyezkednek el, azaz az utolsó paramétertől visszafelé haladva. Ha híváskor elhagyjuk a *default* argumentumokat, akkor *balról-jobbra* haladva kell megadnunk a paraméterek értékeit, de nem hagyhatunk ki paramétert.

Írjunk függvényt, amely két oldal és a közbezárt szög ismeretében kiszámítja a háromszög területét! A szögparaméter alapértelmezésként legyen 90 fok! Aktiváljuk a függvényt kétféle módon! (*DefaultPar*)

Ha a két oldal a derékszögű háromszög befogója, akkor a *default* paraméter megfelelő, ellenkező esetben három paraméterrel kell a függvényt hívni!

```
#include <iostream>
#include <cmath>
using namespace std;
double HaromszogTer(double a, double b, double szog = 90);
int main()
{ double a_oldal, b_oldal, szog = 30, ter1, ter2;
  cout << "a oldal: "; cin >> a_oldal;
  cout << "b oldal: "; cin >> b_oldal;
  cout << "\nDerekszogu haromszog" << endl;
  ter1 = HaromszogTer(a_oldal, b_oldal);
  cout << "terulete: " << ter1 << endl;
  cout << "\nAltalanos haromszog" << endl;
  cout << "ket oldal altal bezart szog: " << szog << endl;
  ter2 = HaromszogTer(a_oldal, b_oldal, szog);
  cout << "terulete: " << ter2;
  cin.get();  cin.get();
  return 0;
}
```

```
double HaromszogTer(double a, double b, double szog)
{
  if (szog == 90)
    return a*b/2;
  else
  {
    double c, s;
    c = sqrt(pow(a,2) + pow(b,2)- 2*a*b*cos(szog*3.141592654/180));
    cout << "c oldal: " << c << endl;
    s = (a + b + c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));  }
}
```

A program futásának eredménye:

```
a oldal: 2
b oldal: 5

Derekszogu haromszog
terulete: 5

Altalanos haromszog
ket oldal altal bezart szog: 30
c oldal: 3.41753
terulete: 2.5
```

6.4.7 Változó hosszúságú paraméterlista

Írjunk programot, amely változó hosszúságú paraméterlista használatával keresi meg az adatok legnagyobb és legkisebb értékét! Használjunk a keresésre kétféle megoldási módszert! (*ValtozoPar*)

```
#include <iostream>
#include <cstdarg>
using namespace std;
int MaxKeres(int elso, ...);
int MinKeres(int n, ...);
int main()
{
  int maxadat, minadat;
  maxadat = MaxKeres(2,6,1,3,-2,8,0);
  cout << "Maximalis adat: " << maxadat << endl;
  minadat = MinKeres(6,2,6,1,3,-2,8);
  cout << "Minimalis adat: " << minadat << endl;
  cin.get();
  return 0;
}
```

A *MaxKeres()* függvény a változó hosszúságú paraméterlista azon hívását alkalmazza, amelyben az adatokat 0-val kell lezárni. A felsorolt adatok közül a függvény a legnagyobb értéket adja vissza a **return** utasításban.

```
int MaxKeres(int elso, ...)
{
  va_list t;
  int  max = elso, kovetkezo;
  va_start(t, elso);
  while (kovetkezo = va_arg(t, int))
  {
     if (kovetkezo > max) max = kovetkezo;
  }
  va_end(t);
  return max;
}
```

A *MinKeres()* függvény a változó hosszúságú paraméterlista azon hívását alkalmazza, amelyben az első adat a paraméterlistán megadott adatok számát jelenti. A felsorolt adatok közül a függvény a legkisebb értékkel tér vissza.

```
int MinKeres(int n, ...)
{
  va_list t;
  int  min, kovetkezo;
  va_start(t, n);
  min = va_arg(t,int);
  for (int i = 1; i<n; i++)
  {
     kovetkezo = va_arg(t,int);
     if (kovetkezo < min) min = kovetkezo;
  }
  va_end(t);
  return min;
}
```

A program futásának eredménye:

```
Maximalis adat: 8
Minimalis adat: -2
```

6.4.8 A main() függvény paraméterei és visszatérési értéke

Írjunk programot, amelyben a ***main()*** függvény paramétereként két szöveget vár! A program jelezze, ha kevés paraméterrel indították! A program kapcsolja össze a két szöveget és jelenítse meg az eredményt! (*MainPar*)

```
#include <iostream>
#include <cstring>
using namespace std;
int main(int argc, char *argv[])
{
 int i;
 char buff1[20], buff2[20], teljes[39];
 if (argc != 3)
 {
   cerr << "Hibas a parameterezes:" << endl;
```

```
    cerr << "A program inditasa: project1 szoveg1 szoveg2" << endl;
    return 1;
  }
  strcpy(buff1, argv[1]); strcpy(buff2, argv[2]);
  cout << "Elso parameter   : " << buff1 << endl;
  cout << "Masodik parameter: " << buff2 << endl;
  cout << endl << "A ket szoveg osszege" << endl;
  strcpy (teljes, buff1);
  strcat (teljes, buff2);
  cout << "Teljes szoveg: " << teljes << endl;
  cin.get();
  return 0;
}
```

A lefordított program indítása:

Project1 paprika malom

A program futásának eredménye:

```
Elso parameter   : paprika
Masodik parameter: malom

A ket szoveg osszege
Teljes szoveg: paprikamalom
```

6.4.9 Rekurzív függvények használata

Írjunk függvényt, amely pozitív, valós alapot pozitív, egész kitevőre emel *rekurzív* módon! (*Hatvany*)

```
#include <iostream>
using namespace std;
double hatvany(double alap, int kitevo);
int main()
{
  double a, ered;
  int k;
  cout << "Hatvanyozas: pozitiv alap pozitiv kitevore" << endl;
  cout << "Alap: "; cin >> a; cout << "Kitevo: "; cin >> k;
  ered = hatvany(a,k);
  if (ered > 0)
   cout << "Hatvany erteke: " << ered << endl;
  cin.get(); cin.get();
  return 0;
}
```

A *hatvany()* függvény az *alap* és *kitevo* paramétereket ellenőrzi, és hibát jelez, ha az aktuális paraméter értéke nem megfelelő. A **switch** utasítás a *kitevo* alapján határozza meg a hatvány értékét. Ha a *kitevo*

0 a hatvány értéke 1,
1 a hatvány értéke az *alap*,

egyébként (**default**) rekurzív módon önmagát hívja a függvény, csökkentve a kitevő értékét.

```
double hatvany(double alap, int kitevo)
{
  if (kitevo < 0 || alap < 0)
  {
    if (kitevo < 0) cerr << "A kitevo negativ!" << endl;
    if (alap < 0) cerr << "Az alap negativ!" << endl;
    return -1;
  }
  switch (kitevo)
  {
    case 0: return 1;
    case 1: return alap;
    default:
      return alap*hatvany(alap,kitevo-1);
  }
}
```

A program futásának eredményei:

```
Hatvanyozas: pozitiv alap pozitiv kitevore
Alap: 5
Kitevo: 3
Hatvany erteke: 125

Hatvanyozas: pozitiv alap pozitiv kitevore
Alap: -2
Kitevo: 3
Az alap negativ!

Hatvanyozas: pozitiv alap pozitiv kitevore
Alap: 3
Kitevo: -2
A kitevo negativ!
```

6.4.10 Inline függvények

Írjunk programot, amely **inline** függvénnyel végzi a fokból radiánba való átváltást! A program két oldalból és a közbezárt szögből számítsa ki a háromszög harmadik oldalát! (*FokRad*)

```
#include <iostream>
using namespace std;
#include <cmath>
inline double fokrad(double fok)
{
   return fok*3.141592654/180;
}
int main()
{ double a, b, c, gamma;
  cout << "A haromszog ket oldala" << endl;
  cout << "A oldal : "; cin >> a;
  cout << "B oldal : "; cin >> b;
  cout << "Kozbezart szog : "; cin >> gamma;
  c = sqrt(pow(a,2)+pow(b,2)-2*a*b*cos(fokrad(gamma)));
  cout << "\nA haromszog harmadik oldala: " << c << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
A haromszog ket oldala
A oldal : 3
B oldal : 4
Kozbezart szog : 30

A haromszog harmadik oldala: 2.05314
```

6.4.11 Függvénynevek átdefiniálása (overloading)

Írjunk programot, amely a kocka felszínét és térfogatát **int** és **double** típusú argumentummal is ki tudja számítani, alkalmazva a függvénynevek átdefiniálását! (*Kocka*)

```
#include <iostream>
using namespace std;
int KockaFelszine(int a);
double KockaFelszine(double a);
int KockaTerfogata(int a);
double KockaTerfogata(double a);
int main()
{
  int a1 = 1;
  double a2 = 1.5;
  cout << "Kocka ele (int)      : " << a1 << endl;
  cout << "Kocka felszine (int) : " << KockaFelszine(a1) << endl;
  cout << "Kocka terfogata (int): " << KockaTerfogata(a1) << endl;
  cout << "\nKocka ele (double)      : " << a2 << endl;
  cout << "Kocka felszine (double) : " << KockaFelszine(a2) << endl;
  cout << "Kocka terfogata (double): " << KockaTerfogata(a2) << endl;
  cin.get();
  return 0;
}

int KockaFelszine(int a)
{
  return 6*a*a;
}

double KockaFelszine(double a)
{
  return 6*a*a;
}
```

```
int KockaTerfogata(int a)
{
   return a*a*a;
}
double KockaTerfogata(double a)
{
   return a*a*a;
}
```

A program futásának eredménye:

```
Kocka ele (int)       : 1
Kocka felszine (int) : 6
Kocka terfogata (int): 1

Kocka ele (double)       : 1.5
Kocka felszine (double) : 13.5
Kocka terfogata (double): 3.375
```

6.4.12 Általánosított függvények (template)

Írjunk programot, amely két vektor skalar szorzatát függvénysablonnal valósítja meg (**template**)! A vektorok adatait konstansként adjuk meg. (*SkalarSzorzat*)

```
#include <iostream>
#include "Fuggv_h.h"
using namespace std;
int a1[] = { 1, 2, 3, 4 };
int b1[] = { 2, 3, 4, 5 };
double x1[] = {1.5, 2.3, 3.2, 4.3 };
double y1[] = {5., 3., 6., 7. };
int main()
{
  int isum;
  double dsum;

  isum = SkalarSzorzat(a1, b1, 4);
  dsum = SkalarSzorzat(x1, y1, 4);
  cout << "1. Skalar szorzat (egesz tipus): " << isum << endl;
  cout << "2. Skalar szorzat (valos tipus): " << dsum << endl;
  cin.get();
  return 0;
}
```

A függvénysablont a *FUGGV_H.H* állományban adjuk meg. Ezt az állományt kell a ***main()*** függvényt tartalmazó állományába beszerkeszteni "" (aposztrofok) között, mivel saját .H fájl abban a könyvtárban található, ahol a program állománya is elhelyezkedik.)

```
#include "Fuggv_h.h"
```

```
//FUGGV_H.H
template <class TIPUS>
TIPUS SkalarSzorzat (TIPUS a[], TIPUS b[], int n)
{
  TIPUS s = 0;
  for (int i = 0; i<n; i++)
  {
     s +=a[i] * b[i];
  }
  return s;
}
```

A program futásának eredménye:

```
1. Skalar szorzat (egesz tipus): 40
2. Skalar szorzat (valos tipus): 63.7
```

Írjunk programot, amely a kocka felszínének és térfogatának számítását sablon alkalmazásával oldja meg! (*KockaSablon*)

```
#include <iostream>
#include "Fuggv.h"
using namespace std;

int main()
{
  int a1 = 1;
  double a2 = 1.5;
  cout << "Kocka ele (int)       : " << a1 << endl;
  cout << "Kocka felszine (int) : "
       << KockaFelszine(a1) << endl;
  cout << "Kocka terfogata (int): "
       << KockaTerfogata(a1) << endl;
  cout << "\nKocka ele (double)       : " << a2 << endl;
  cout << "Kocka felszine (double) : "
       << KockaFelszine(a2) << endl;
  cout << "Kocka terfogata (double): "
       << KockaTerfogata(a2) << endl;
  cin.get();
  return 0;
}
```

A *Fugg.h* állomány tartalmazza a **template** függvényeket a kocka felszínének és térfogatának különböző típussal történő számításához.

```
//FUGGV.H
template <class T>
T KockaFelszine(T a)
{
  return 6*a*a;
}
```

```
template <class T>
T KockaTerfogata(T a)
{
   return a*a*a;
}
```

A program futásának eredménye:

```
Kocka ele (int)       : 1
Kocka felszine (int) : 6
Kocka terfogata (int): 1

Kocka ele (double)       : 1.5
Kocka felszine (double) : 13.5
Kocka terfogata (double): 3.375
```

7. Névterek és modulok használata

Ebben a fejezetben néhány példán keresztül bemutatjuk a különböző névtér (névterület - **namespace**), az **extern** tárolási osztályú változók és függvények, valamint a modulok (több állományból felépülő alkalmazás) használatát.

7.1 Saját névterek kialakítása és használata

Írjunk programot, amelyben *Olvas()*függvénnyel koordináta-párokat olvashatunk be, amelyeket a *Rendez()* függvénnyel sorba rendezhetünk az *x*-koordináta szerint, és a *Kiir()* függvénnyel kiírathatjuk a rendezett koordináta-párokat! A függvények számára tervezzünk *kozos* névteret (**namespace**)! Írjunk főfüggvényt, amely bemutatja a függvények használatát! (*Nevter1*)

A *KOZOS.H* állomány a függvények prototípusát tartalmazza.

```
//KOZOS.H
namespace kozos
{
  void Olvas(double x[], double y[], int n);
  void Rendez(double x[], double y[], int n);
  void Kiir(double x[], double y[], int n);
}
```

A *KOZOS.CPP* állományban kidolgozzuk a függvényeket, a :: (hatókör) operátorral jelezve, hogy melyik névtérhez tartoznak.

```
//KOZOS.CPP
#include <iostream>
#include <iomanip>
using namespace std;
#include "kozos.h"
using namespace kozos;
void kozos::Olvas(double x[], double y[], int n)
{
   for (int i = 0; i< n; i++)
   {
      cout << endl << "x[" <<i << "] = "; cin >> x[i];
      cout << "y[" <<i << "] = "; cin >> y[i];
   }
}
void kozos::Rendez(double x[], double y[], int n)
{
  double sx, sy;
  for (int i = 0; i< n-1; i++)
  {
    for (int j = i+1; j<n; j++)
    {
      if (x[j] < x[i])
```

```
        {
          sx = x[i]; x[i] = x[j]; x[j] = sx;
          sy = y[i]; y[i] = y[j]; y[j] = sy;
        }
      }
  }
}
void kozos::Kiir(double x[], double y[], int n)
{
  cout << endl << "Rendezett adatok" << endl;
  cout << setw(10) << "x" << setw(10) << "y" << endl;
  cout << "--------------------\n";
  for (int i = 0; i<n; i++)
  {
    cout << setw(10) << x[i] << setw(10) << y[i] << endl;
  }
}
```

A *MAIN.CPP* állomány tartalmazza a főfüggvényt, amely *include* fordító direktívával éri el a *KOZOS.H* állományt, valamint a **using namespace** `kozos;` jelzi, hogy a névtérből mindenhez hozzá akar férni.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "kozos.h"
using namespace kozos;
int main()
{
  double *xx, *yy, xx1, yy1;
  int db;
  cout << "Elemek szama: "; cin >> db;
  xx = new double[db];
  yy = new double[db];
  Olvas(xx,yy,db);
  Rendez(xx,yy,db);
  Kiir(xx,yy,db);
  delete [] xx;  delete [] yy;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 4

x[0] = 3
y[0] = 12.5

x[1] = 2
y[1] = 11.6

x[2] = 6
y[2] = 15.6
```

```
x[3] = 5
y[3] = 12.7

Rendezett adatok
         x         y
--------------------
         2      11.6
         3      12.5
         5      12.7
         6      15.6
```

Írjunk programot, amelyben a *kozos* névtér egyes elemeihez akarunk hozzáférni, csak az *Olvas()* és a *Kiir()* függvényeket kívánjuk használjuk! (*Nevter2*)

Ilyen esetben a **using** kulcsszó után áll a névtér neve, melyet a hatókör operátorral (**::**) elválasztva követ a felhasználni kívánt függvény neve, majd pontosvessző zárja az utasítást. Ezt minden függvény esetén meg kell ismételni.

```
#include <iostream>
using namespace std;
#include "kozos.h"
using kozos::Olvas;
using kozos::Kiir;
int main()
{
  double *xx, *yy, xx1, yy1;
  int db;
  cout << "Elemek szama: "; cin >> db;
  xx = new double[db];
  yy = new double[db];
  Olvas(xx,yy,db);
  Kiir(xx,yy,db);
  delete [] xx;
  delete [] yy;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 3                      Tombok listaja
                                            x         y
x[0] = 1.2                           --------------------
y[0] = 12.567                               1.2    12.567
                                            2.3    23.567
x[1] = 2.3                                  5.2    34.123
y[1] = 23.567

x[2] = 5.2
y[2] = 34.123
```

7.2 Az extern tárolási osztály

Írjunk programot, amelyben a függvényekhez **extern** tárolási osztállyal férünk hozzá! (*Extern*)

A függvények a *kozos.cpp* állomány tárolja:

```
//KOZOS.CPP
#include <iostream>
#include <iomanip>
using namespace std;
```

```
void Olvas(double x[], double y[], int n)
{
   for (int i = 0; i< n; i++)
   {
      cout << endl << "x[" <<i << "] = "; cin >> x[i];
      cout << "y[" <<i << "] = "; cin >> y[i];
   }
}

void Rendez(double x[], double y[], int n)
{
  double sx, sy;
  for (int i = 0; i< n-1; i++)
  {
    for (int j = i+1; j<n; j++)
    {
      if (x[j] < x[i])
      {
        sx = x[i]; x[i] = x[j]; x[j] = sx;
        sy = y[i]; y[i] = y[j]; y[j] = sy;
      }
    }
  }
}

void Kiir(double x[], double y[], int n)
{
  cout << endl << "Rendezett adatok" << endl;
  cout << setw(10) << "x" << setw(10) << "y" << endl;
  cout << "--------------------\n";
  for (int i = 0; i<n; i++)
  {
    cout << setw(10) << x[i] << setw(10) << y[i] << endl;
  }
}
```

A főfüggvény előtt pedig az **extern** tárolási osztállyal kell megjelölnünk a használni kívánt függvényeket.

```
//MAIN.CPP
#include <iostream>
using namespace std;

extern void Olvas(double x[], double y[], int n);
extern void Rendez(double x[], double y[], int n);
extern void Kiir(double x[], double y[], int n);
int main()
{
  double *xx, *yy, xx1, yy1;
  int db;
  cout << "Elemek szama: "; cin >> db;
  xx = new double[db];
  yy = new double[db];
  Olvas(xx,yy,db);
  Rendez(xx,yy,db);
```

```
   Kiir(xx,yy,db);
   delete [] xx;
   delete [] yy;
   cin.get(); cin.get();
   return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 3

x[0] = 1.2
y[0] = 4.67

x[1] = 2.6
y[1] = 6.78

x[2] = 3.6
y[2] = 7.12

Rendezett adatok
         x         y
--------------------
       1.2      4.67
       2.6      6.78
       3.6      7.12
```

7.3 Több modulból álló C++ program

Írjunk programot, amelyben több modulban tárolt függvényekhez férünk hozzá a .h állomány használatával! (*Modulok*)

A függvények prototípusát egy deklarációs állományban kell megadni.

```
//KOZOS.H
void Olvas(double x[], double y[], int n);
void Rendez(double x[], double y[], int n);
void Kiir(double x[], double y[], int n);
```

A függvényeket egy C++ forrásfájlban kell kidolgozni:

```
//KOZOS.CPP
#include <iostream>
#include <iomanip>
using namespace std;

void Olvas(double x[], double y[], int n)
{
   for (int i = 0; i< n; i++)
   {
      cout << endl << "x[" <<i << "] = "; cin >> x[i];
      cout << "y[" <<i << "] = "; cin >> y[i];
   }
}
```

```
void Rendez(double x[], double y[], int n)
{
  double sx, sy;
  for (int i = 0; i< n-1; i++)
  {
    for (int j = i+1; j<n; j++)
    {
      if (x[j] < x[i])
      {
        sx = x[i]; x[i] = x[j]; x[j] = sx;
        sy = y[i]; y[i] = y[j]; y[j] = sy;
      }
    }
  }
}

void Kiir(double x[], double y[], int n)
{
  cout << endl << "Rendezett adatok" << endl;
  cout << setw(10) << "x" << setw(10) << "y" << endl;
  cout << "--------------------\n";
  for (int i = 0; i<n; i++)
  {
    cout << setw(10) << x[i] << setw(10) << y[i] << endl;
  }
}

//MAIN.CPP
#include "Kozos.h"
#include <iostream>
using namespace std;
int main()
{
  double *xx, *yy, xx1, yy1;
  int db;
  cout << "Elemek szama: "; cin >> db;
  xx = new double[db];
  yy = new double[db];
  Olvas(xx,yy,db);
  Rendez(xx,yy,db);
  Kiir(xx,yy,db);
  delete [] xx;
  delete [] yy;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 2                    Rendezett adatok
                                            x         y
x[0] = 3                           --------------------
y[0] = 12                                   1        45
x[1] = 1                                    3        12
y[1] = 45
```

Írjunk programot, amely a kocka felszínét és térfogatát számító függvényeket önálló forrásállományokban definiálja, de közös projektben. (*Modulok2*)

```
//MAIN.CPP
#include <iostream>
#include "Fuggv1.h"
#include "Fuggv2.h"
using namespace std;

int main()
{
  int a1 = 1;
  double a2 = 1.5;
  cout << "Kocka ele (int)       : " << a1 << endl;
  cout << "Kocka felszine (int) : "
       << KockaFelszine(a1) << endl;
  cout << "Kocka terfogata (int): "
       << KockaTerfogata(a1) << endl;
  cout << "\nKocka ele (double)       : " << a2 << endl;
  cout << "Kocka felszine (double) : "
       << KockaFelszine(a2) << endl;
  cout << "Kocka terfogata (double): "
       << KockaTerfogata(a2) << endl;
  cin.get();
  return 0;
}
```

A *FUGGV1.H* és a *FUGGV2.H* fejlécállományok tartalmazzák a függvények prototípusát.

```
//FUGGV1.H
int KockaFelszine(int a);
int KockaTerfogata(int a);

//FUGGV2.H
double KockaFelszine(double a);
double KockaTerfogata(double a);
```

A *FUGGV1.CPP* állomány pedig a függvények definícióit tartalmazza.

```
//FUGGV1.CPP
int KockaFelszine(int a)
{
  return 6*a*a;
}
int KockaTerfogata(int a)
{
  return a*a*a;
}
```

```
//FUGGV2.CPP
double KockaFelszine(double a)
{
  return 6*a*a;
}
double KockaTerfogata(doule a)
{
  return a*a*a;
}
```

A program futásának eredménye:

```
Kocka ele (int)       : 1
Kocka felszine (int) : 6
Kocka terfogata (int): 1

Kocka ele (double)       : 1.5
Kocka felszine (double) : 13.5
Kocka terfogata (double): 3.375
```

8. Algoritmusok programozása

Ebben a fejezetben algoritmusok programozására mutatunk be néhány példát: osztók, legnagyobb közös osztó, legkisebb közös többszörös keresése. Foglalkozunk különféle rendezési algoritmusokkal, interpolációkkal.

8.1 Osztók, legnagyobb közös osztó, legkisebb közös többszörös keresése

Olvassunk be egy egész számot, és írjuk ki a szám összes osztóját! A feladatot oldjuk meg **for**, **while** és **do-while** ciklusok használatával is! (*Alg_01*)

```
//KOZOS.H
#include <iostream>
#include <iomanip>
using namespace std;
void oszto1(long int szam);
void oszto2(long int szam);
void oszto3(long int szam);
```

```
//KOZOS.CPP
#include "Kozos.h"
void oszto1(long int szam)
{
  long int oszto, fele;
  fele = szam /2;
  cout << "Osztoi:" << endl;
  for (oszto = 2; oszto <= fele; oszto++)
  {
    if (szam % oszto == 0) cout << setw(6) << oszto << endl;
  }
}
void oszto2(long int szam)
{
  long int oszto, fele;
  fele = szam /2;
  cout << "Osztoi:" << endl;
  oszto = 2;
  while (oszto <= fele)
  {
    if (szam % oszto == 0) cout << setw(6) << oszto << endl;
    oszto++;
  }
}
void oszto3(long int szam)
{
  long int oszto, fele;
  fele = szam /2;
  cout << "Osztoi:" << endl;
  oszto = 2;
  do
  {
    if (szam % oszto == 0) cout << setw(6) << oszto << endl;
    oszto++;
  } while(oszto <= fele);
}

//MAIN.CPP
#include "Kozos.h"
int main()
{
  int x;
  cout << "Szam: "; cin >> x;
  cout << "for ciklussal" << endl;
  oszto1(x);
  cin.get();cin.get();
  cout << "while ciklussal" << endl;
  oszto2(x);
  cin.get();
  cout << "do - while ciklussal" << endl;
  oszto3(x);
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Szam: 24                while ciklussal         do - while ciklussal
for ciklussal           Osztoi:                 Osztoi:
Osztoi:                      2                       2
     2                       3                       3
     3                       4                       4
     4                       6                       6
     6                       8                       8
     8                      12                      12
    12
```

Olvassunk be egy egész számot, határozzuk meg a szám törzstényezőit, és hatványkitevős alakban is jelenítsük meg azokat!(*Alg_02*)

```
//KOZOS.H
#include <iostream>
#include <iomanip>
using namespace std;
void torzstenyezok(int szam);
void hatvanykitevos(int szam);

//KOZOS.CPP
#include "Kozos.h"
void torzstenyezok(int szam) // a szam törzsténzyzőkre való bontása
{
  int oszto;
  if (szam != 1 && szam > 0)
  {
    cout << "\nTorzstenyezok:" << endl;
    oszto = 2;
    while (szam != 1 && szam > 0)
    {
      while (szam %oszto != 0)
      {
        oszto++;
      }
      szam /= oszto;
      cout << oszto;
      if (szam != 1) cout << " * ";
    }
  }
}
// a szam felbontása hatványkitevős alakra
void hatvanykitevos(int szam)
{
  int oszto = 2, hatv = 1, elozo = 1;
  if (szam != 1 && szam > 0)
  {
    cout << endl << "\nHatvanykitevos alak:" << endl;
    while (szam != 1 && szam > 0)
    {
      while (szam % oszto != 0)
      {
        oszto++;
      }
      szam /= oszto;
      if (oszto == elozo)
        hatv++;
      else
      {
        if (elozo != 1)
        {
          if (hatv > 1)
            cout << elozo << "^" << hatv << "*";
          else
            cout << elozo << "*";
        }
        hatv = 1;
      }
      elozo = oszto;
      if (szam == 1)
       if (hatv > 1)
         cout << oszto << "^" << hatv;
       else
         cout << oszto;
    }
  }
}
```

```
//MAIN.CPP
#include "Kozos.h"
int main()
{
  int x;
  cout << "Szam: "; cin >> x;
  torzstenyezok(x);
  hatvanykitevos(x);
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Szam: 1250

Torzstenyezok:
2 * 5 * 5 * 5 * 5

Hatvanykitevos alak:
2*5^4
```

Olvassunk be két pozitív egész számot, határozzuk meg a legnagyobb közös osztójukat és a legkisebb közös többszörösüket! (*Alg_03*)

A legnagyobb közös osztót és a legkisebb közös többszöröst háromféle módszerrel mutatjuk be:

- A *végigpróbál* módszer. A kisebb számig nézi meg 2-től mindkét szám közös osztóit.
- Az *euklideszi* algoritmus. Az euklideszi algoritmus szerint, ha a kisebbik szám maradék nélkül van meg a nagyobbikban, a kisebbik lesz a legnagyobb közös osztó. Ha van maradék, a legnagyobb közös osztó kisebb vagy egyenlő lesz a maradékkal. A következő próbálkozásnál a kisebbik szám lép az osztandó, a maradék pedig az osztó helyébe. Az algoritmus így mindaddig folytatódik, míg a maradék nulla nem lesz, ekkor az aktuális osztó lesz a legnagyobb közös osztó.
- A *törzstényezőkre* bontás módszer. A legkisebb közös többszörös meghatározásánál törzstényezőkre bontjuk a számokat. Először az *szam1*-nek keressük a törzstényezőit, a megtalált törzstényezővel elosztjuk a *szam1*-t. A *szám2*-őt csak akkor osztjuk el, ha osztója a számnak. A legkisebb többszöröst beszorozzuk a megtalált törzstényezővel, ezt addig végezzük, míg a *szam1* 1 lesz. Ezután a *szam2*-vel is addig folytatjuk a törzstényezőkre bontást, míg a *szam2* is 1-gyé válik.

```
//KOZOS.H
#include <iostream>
#include <iomanip>
using namespace std;
void Vegigprobal(int szam1, int szam2);
void Euklideszi(int szam1, int szam2);
void Torzstenyezok_lnko(int szam1, int szam2);

//KOZOS.CPP
#include "Kozos.h"
void Vegigprobal(int szam1, int szam2)
{
  int eddig, oszto, kozos =1, v;
  cout << endl << "Vegigprobal modszerrel" << endl << endl;
  if (szam1 > 0 && szam2 > 0)
  {
     eddig = szam1;
     if (szam2 < eddig)
        eddig = szam2;
    for (oszto = 2; oszto <= eddig; oszto++)
    {
      if (szam1 % oszto == 0 && szam2 % oszto == 0)
         kozos = oszto;
    }
    cout << "Legnagyobb kozos oszto: " << kozos << endl;
    v = szam1*szam2/kozos;
    cout << "Legkisebb kozos tobbszoros: " << v << endl;
  }
}
void Euklideszi(int szam1, int szam2)
{
  int maradek, temp, s1, s2, v;
  cout << endl << "Euklideszi modszerrel" << endl << endl;
  if (szam1 > 0 && szam2 > 0)
  {
    s1 = szam1; s2 = szam2;
    if (szam1 < szam2)
    {
      temp = szam1; szam1 = szam2; szam2 = temp;
    }
    maradek = szam2;
    while (maradek != 0)
    {
      maradek = szam1 % szam2;
      szam1 = szam2;
      szam2 = maradek;
    }
    cout << "Legnagyobb kozos oszto: " << szam1 << endl;
    v = s1 * s2 / szam1;
    cout << "Legkisebb kozos tobbszoros: " << v << endl;
  }
}
void Torzstenyezok_lnko(int szam1, int szam2)
{
  int s1, s2, lktobb, oszto1, oszto2, v;
  cout << endl <<"Torzstenyezokre bontas modszerrel"
       << endl << endl;
  if (szam1 > 0 && szam2 > 0)
  {
    s1 = szam1;
    s2 = szam2;
    lktobb = 1;
    while (szam2 != 1 || szam1 != 1)
    {
      oszto1 = 2;
      oszto2 = 2;
      while (szam1 != 1)
      {
        while (szam1 % oszto1 != 0)
        {
          oszto1++;
        }
        if(szam2 % oszto1 == 0)
          szam2 /= oszto1;
        lktobb*= oszto1;
        szam1 /= oszto1;
      }
      while (szam2 != 1)
      {
        while (szam2 % oszto2 != 0)
        {
          oszto2++;
        }
        szam2 /=oszto2;
        lktobb*=oszto2;
      }
    }
    v = s1*s2/lktobb;
    cout << "Legnagyobb kozos oszto: " << v << endl;
    cout << "Legkisebb kozos tobbszoros: " << lktobb << endl;
  }
 }

//MAIN.CPP
#include "Kozos.h"
int main()
{
  int x, y;
  cout << "Szam1: "; cin >> x;
  cout << "Szam2: "; cin >> y;
  Vegigprobal(x,y);
  Euklideszi(x,y);
  Torzstenyezok_lnko(x,y);
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Szam1: 30
Szam2: 24

Vegigprobal modszerrel

Legnagyobb kozos oszto: 6
Legkisebb kozos tobbszoros: 120
```

```
Euklideszi modszerrel

Legnagyobb kozos oszto: 6
Legkisebb kozos tobbszoros: 120

Torzstenyezokre bontas modszerrel

Legnagyobb kozos oszto: 6
Legkisebb kozos tobbszoros: 120
```

8.2 Rendezési algoritmusok

Rendezzük valós számokat a csere, a közvetlen beszúrás és a buborék módszerrel! (*Alg_04*)

```
//KOZOS.H
#include <iostream>
#include <iomanip>
using namespace std;
void Csere(double x[], int m);
void Kozvetlen(double x[], int m);
void Buborek(double x[], int m);
void Kiir(double x[], int m);
```

A valós számok rendezésére háromféle módszert mutatunk be:

- *Rendezés cserével.* Legyen az adatok száma *m*, és a feladat az adatok növekvő sorrendbe rendezése. A rögzített *i*-edik elemet összehasonlítjuk a nála nagyobb indexű elemekkel. Ha a vizsgált elem kisebb az *i*-edik elemnél, az elemeket egymással cseréljük fel. A csere algoritmust két egymásba ágyazott ciklussal valósítjuk meg.
- *Rendezés közvetlen beszúrással.* A közvetlen beszúrással történő rendezés esetén az algoritmus úgy működik, hogy a 0 indexű elemtől kezdve minden lépésben kiemeljük a sorozat előző (kisebb) elemét és beszúrjuk a megfelelő helyre.
- *Buborék rendezés.* A rendezés elnevezése abból ered, hogy a rendezendő adatokat a víztartályban lebegő buborékok súlyának képzeljük el, és a rendezés minden lépésben a még nem rendezett legkisebb buborék felemelkedik a súlyának megfelelő szintre.

```
//KOZOS.CPP
#include "Kozos.h"
void Csere(double x[], int m)
{ double s;
  for (int i=0; i< m-1; i++)
  {
    for (int j=i+1; j<m; j++)
    {
      if (x[j] < x[i])
      { s = x[i];  x[i] = x[j]; x[j] = s; }
    }
  }
}
void Kozvetlen(double x[], int m)
{
  int j;
  double buf;
  for (int i = 1; i< m; i++)
  {
    buf = x[i]; j = i-1;
    if (j>= 0)
    {
      while (buf < x[j])
      {
        x[j+1] = x[j]; j--;
        if (j <0) break;
      }
    }
   x[j+1] = buf;
  }
}
void Buborek(double x[], int m)
{
  double buf;
  for (int i = 1; i<m; i++)
  {
    for (int j = m-1; j>=i; j--)
    {
      if (x[j - 1] > x[j])
      {
        buf = x[j - 1]; x[j-1] = x[j]; x[j] = buf;
      }
    }
  }
}
void Kiir(double x[], int m)
{
  int db=0;
  for (int i = 0; i<m; i++)
  {
    cout << setw(8) << x[i]; db++;
    if (db % 8 == 0) { db = 0; cout << endl; }
  }
}
//MAIN.CPP
#include "Kozos.h"
double x1[] = {5,1,3,7,2,8,4,6};
double x2[] = {5,1,3,7,2,8,4,6};
double x3[] = {5,1,3,7,2,8,4,6};
int main()
{ int db;
  db = sizeof x1/sizeof(double);
  cout << "Rendezes Csere modszerrel" << endl << endl;
  Csere(x1,db);
  Kiir(x1,db);
  db = sizeof x2/sizeof(double);
  cout << "Rendezes Kozvetlen modszerrel" << endl << endl;
  Kozvetlen(x2,db);
  Kiir(x2,db);
  db = sizeof x3/sizeof(double);
  cout << "Rendezes Buborek modszerrel" << endl << endl;
  Buborek(x3,db);
  Kiir(x3,db);
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Rendezes Csere modszerrel
      1       2       3       4       5       6       7       8
Rendezes Kozvetlen modszerrel
      1       2       3       4       5       6       7       8
Rendezes Buborek modszerrel
      1       2       3       4       5       6       7       8
```

8.3 Interpolációk

Interpoláció: ha néhány pontjával megadott függvényt kell folytonos görbével ábrázolnunk, a függvény ismeretlen pontjait mindig interpolációs módszerrel határozzuk meg.

Kétféle interpolációs módszert mutatunk be:

- *Lineáris interpoláció*. A legegyszerűbb (legkisebb fokszámú) közelítés a lineáris interpoláció. Ismert az $(x_i, f(x_i))$, illetve $(x_{i+1}, f(x_{i+1}))$ pontpár, keressük x_1 helyen $f(x_1)$ értékét egyenessel közelítve.
- *Lagrange interpoláció*. A két alappontot (x_0, x_1) felhasználó közelítés $(n=1)$ a két pont közötti egyenessel közelít. Belátható, hogy a három alappontos közelítés $(n=2)$ egy másodfokú függvény, parabola lesz. A négy alappontos közelítés harmadfokú függvény, és így tovább, $n+1$ alappont esetén bebizonyítható, hogy a közelítés n-edfokú polinom lesz. A polinom együtthatóinak kiszámítására Lagrange dolgozott ki eljárást.

Olvassunk be alappontokat *(x,y)*, és a kívánt *xx1* helyen számítsuk ki az interpolált értéket *lineáris* interpoláció felhasználásával! *(Alg_05)*

```
//KOZOS.H
void Olvas(double x[], double y[], int n);
void Rendez(double x[], double y[], int n);
void Kiir(double x[], double y[], int n);

//KOZOS.CPP
#include <iostream>
#include <iomanip>
using namespace std;
void Olvas(double x[], double y[], int n)
{
   for (int i = 0; i< n; i++)
   {
      cout << endl << "x[" <<i << "] = "; cin >> x[i];
      cout << "y[" <<i << "] = "; cin >> y[i];
   }
}
```

```
void Rendez(double x[], double y[], int n)
{
  double sx, sy;
  for (int i = 0; i< n; i++)
  {
    for (int j = i+1; j<n-1; j++)
    {
      if (x[j] < x[i])
      {
        sx = x[i]; x[i] = x[j]; x[j] = sx;
        sy = y[i]; y[i] = y[j]; y[j] = sy;
      }
    }
  }
}
void Kiir(double x[], double y[], int n)
{
  cout << endl << "Rendezett adatok" << endl;
  for (int i = 0; i<n; i++)
  {
    cout << setw(10) << x[i] << setw(10) << y[i] << endl;
  }
}

//INTPOL.H
int Linint(double *x, double *y, int n, double x1, double& y1);

//INTPOL.CPP
#include <iostream>
using namespace std;
int Linint(double *x, double *y, int n, double x1, double& y1)
{
  for (int i = 0; i<n-1; i++)
  {
    if (x[i] <= x1 && x[i+1] >= x1)
    {
      if (x[i+1] == x[i])
      {
        cerr << "Hibas adat" << endl;
        return 1;
      }
      y1 = y[i] +(y[i+1]-y[i])/(x[i+1]-x[i])*(x1-x[i]);
      return 0;
    }
  }
  return 1;
}

//MAIN.CPP
#include "Intpol.h"
#include "Kozos.h"
#include <iostream>
using namespace std;
```

```
int main()
{
  double *xx, *yy, xx1, yy1;
  int db;
  cout << "Elemek szama: "; cin >> db;
  xx = new double[db]; yy = new double[db];
  Olvas(xx,yy,db);
  Rendez(xx,yy,db);
  Kiir(xx,yy,db);
  cout <<  endl << "Linearis interpolacio" << endl;
  cout << "Interpolalando ertek: "; cin >> xx1;
  if(Linint(xx,yy,db,xx1,yy1)==0)
    cout << "Interpolalt ertek   : " << yy1 << endl;
  else
    cerr << "Hibas ertek!" << endl;
  cin.get(); cin.get();
  delete [] xx; delete [] yy;
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 4

x[0] = 1
y[0] = 1

x[1] = 2
y[1] = 1.8

x[2] = 3
y[2] = 2.2

x[3] = 4
y[3] = 2.5
```

```
Rendezett adatok
            1        1
            2      1.8
            3      2.2
            4      2.5

Linearis interpolacio
Interpolalando ertek: 2.5
Interpolalt ertek   : 2
```

Olvassunk be alappontokat (*x,y*), és a kívánt *xx1* helyen számítsuk ki az interpolált értéket *Lagrange* interpoláció felhasználásával!(*Alg_06*)

```
int Lagrange(double *x, double *y, int n, double x1, double& y1)
{
  double p = 0.0, s;
  for (int i = 0; i< n; i++)
  { s = 1.0;
    for (int j = 0; j<n; j++)
    {
      if (i != j)
      {
        if (x[i] == x[j])
        {
          cerr << "Hibas adat" << endl; return 1;
        }
        s *=(x1-x[j])/(x[i] - x[j]);
      }
    }
  p += y[i]*s;
  }
  y1= p;
  return 0;
}

//MAIN.CPP
#include "Intpol.h"
#include "Kozos.h"
#include <iostream>
using namespace std;
int main()
{
  double *xx, *yy, xx1, yy1;
  int db;
  cout << "Elemek szama: "; cin >> db;
  xx = new double[db];
  yy = new double[db];
  Olvas(xx,yy,db);
  Rendez(xx,yy,db);
  Kiir(xx,yy,db);
  cout <<  endl << "Lagrange interpolacio" << endl;
  cout << "Interpolalando ertek: "; cin >> xx1;
  if (Lagrange(xx,yy,db,xx1,yy1)==0)
    cout << "Interpolalt ertek   : " << yy1 << endl;
  else
    cerr << "Hibas ertek!" << endl;
  cin.get(); cin.get();
  delete [] xx;
  delete [] yy;
  return 0;
}
```

A program futásának eredménye:

```
Elemek szama: 4                     Rendezett adatok
                                              1         1
x[0] = 1                                      2       1.8
y[0] = 1                                      3       2.2
                                              4       2.5
x[1] = 2
y[1] = 1.8
                                    Lagrange interpolacio
x[2] = 3                            Interpolalando ertek: 2.5
y[2] = 2.2                          Interpolalt ertek   : 2.0312533

x[3] = 4
y[3] = 2.5
```

9. Láncolt lista

Ebben a fejezetben egyszerű példán mutatjuk be az előre és a hátra láncoló lista készítését.

9.1 Előre-hátra láncolt lista

Írjunk programot, amely a *Pont* struktúrában tárolt koordinátapont adatait kétirányú láncba fűzi! Írjunk függvényeket, amelyek beolvassák az adatokat, elkészítik a láncot, kiírják a lánc tartalmát elölről és visszafelé, beszúrnak egy új elemet a lánc végére, valamint törölnek egy elemet a láncból!(*Lista*)

A *LISTA.H* állomány tartalmazza a *Pont* struktúra definícióját és a függvények prototípusát:

```
//LISTA.H
struct Pont {
               int x,y;
               Pont *kov, *elozo;
             };

Pont* Olvas(Pont *&u);
void KiirElore(Pont* e);
void KiirHatra(Pont* u);
Pont* BeszurV(Pont *v, int x1, int y1);
Pont* Torol(Pont *e, Pont *&u, int x1, int y1);
```

A *LISTAFG.CPP* állomány tartalmazza a függvényeket:

- Az *Olvas()* függvény prototípusa:

  ```
  Pont* Olvas(Pont *&u)
  ```

  Az *Olvas()* függvény értékként adja vissza a lánc elejére mutató első pointert, és a paraméterlistán a lánc végére mutató pointert. Így könnyen hozzáférhetünk elölről, illetve hátulról a lánc elmeihez.

  Az *Olvas()* függvény beolvassa az adatok darabszámát, dinamikusan létrehozza a lánc elemeit a **new** utasítással, majd beolvassa az (*x*,*y*) koordinátapontot, nullázza az *elozo* és a *kov* mutatókat, illetve láncba fűzi az elemeket. Figyeli, hogy volt-e már eleme a láncnak, mert ha még nem volt, akkor ez lesz az *elso*. A továbbiakban mindig a lánc végére fűzi az újabb elemeket.

- A *KiirElore()* függvény a paraméterlistán kapja a lánc első elemére mutató pointert. A függvény prototípusa:

  ```
  void KiirElore(Pont* e)
  ```

A **while** ciklus figyeli, a lánc végét, amelyet a 0 értékű mutató jelez:

```
while (e != 0)
```

A ciklusban tovább kell lépnünk a következő elemre mutató pointerre, különben végtelen ciklust kapunk:

```
e = e->kov;
```

- A *KiirHatra()* függvény a paraméterlistán kapja a lánc utolsó elemére mutató pointert. A függvény prototípusa:

  ```
  void KiirHatra(Pont* u)
  ```

  Ebben az esetben a visszamutató pointert kell ez előző elemre állítani a ciklus leállítása miatt:

  ```
  u = u->elozo;
  ```

- A *BeszurV()* függvény paraméterlistán kapja a lánc első elemére mutató pointert és a beszúrandó (*x,y*) koordináta adatait a lánc végére szúrja·be. Visszatérési értékként a lánc elejére mutató, *elso* pointer megváltozott értékét szolgáltatja.

  ```
  Pont* BeszurV(Pont* v, int x1, int y1)
  ```

- A *Torol()* függvény a paraméterlistán kapja a lánc *elso* és az *utolso* elemére mutató pointert, valamint a törlésre szánt koordinátapont (*x1,y1*) adatát. Az *utolso* mutató megváltozott értékét pointerben adja vissza:

  ```
  Pont* Torol(Pont *elso, Pont *&utolso, int x1, int y1)
  ```

```
//LISTAFG
#include <iostream>
#include <iomanip>
using namespace std;
#include "Lista.h"
Pont* Olvas(Pont *&u)
{
 int db;
 Pont *uj, *elso = 0, *utolso = 0;
 cout << "Adatok szama: ";
 cin >> db;
 for (int i = 0; i< db; i++)
 {
   uj = new Pont;
   cout << "\nx: "; cin >> uj->x;
   cout << "y: "; cin >> uj->y;
   uj->kov = 0; uj->elozo = 0;
   if (elso == 0) elso = uj;
   else
   {
     utolso->kov = uj; uj->elozo = utolso;
   }
   utolso = uj;
 }
 u = utolso;
 return elso;
}
```

```
void KiirElore(Pont* e)
{ cout << "\nA lancolt lista elemei (elore)\n";
  cout << setw(5) << "X" << setw(5) << "Y" << endl;
  while (e != 0)
  {
    cout << setw(5) << e->x << setw(5) << e->y << endl;
    e = e->kov;
  }
}
void KiirHatra(Pont* u)
{ cout << "\nA lancolt lista elemei (visszafele)\n";
  cout << setw(5) << "X" << setw(5) << "Y" << endl;
  while (u != 0)
  {
    cout << setw(5) << u->x << setw(5) << u->y << endl;
    u = u->elozo;
  }
}
Pont* BeszurV(Pont* v, int x1, int y1)
{  Pont *uj;
   if (!v) return v; // csak létező listába szúr be
   uj= new Pont;
   uj->x = x1;
   uj->y = y1;
   uj->kov = 0;
   v->kov = uj;
   uj->elozo = v;
   v = uj;
   return v;
}
Pont* Torol(Pont *elso, Pont *&utolso, int x1, int y1)
{ Pont *p;
  int talalt = 0;
  p = elso;
  while (p!= 0)
  {
    if (p->x == x1 && p->y == y1)
    {
      talalt++;
      if (p->elozo == 0 && p->kov == 0)
      { elso = 0; utolso = 0; delete p; break; }
      if (p->elozo != 0) (p->elozo)->kov = p->kov;
      else { elso = p->kov; (p->kov)->elozo = 0; }
      if (p->kov != 0) (p->kov)->elozo = p->elozo;
      else { utolso = p->elozo; (p->elozo)->kov = 0; }
      delete p; break;
    }
    p = p->kov;
  }
  if (talalt == 0)
    cerr << "\nHibas adat: (" << x1 << "," << y1
         << ") nincs a lancban\n";
  return elso;
}
```

A ***main()*** függvény bemutatja a *Pont* struktúrából felépített lánc módosítását, új elem beszúrását, listázását és törlését.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Lista.h"
int main()
{
  int x1, y1;
  Pont *pelso, *putolso, *p, *p1;
  pelso = Olvas(putolso);
  KiirElore(pelso);
  putolso = BeszurV(putolso,5,6);  // beszúrás a láncba
  putolso = BeszurV(putolso,11,7); // beszúrás a láncba
  KiirElore(pelso);
  cout << "\nPont torlese\n";
  cout << " x: "; cin >> x1;
  cout << " y: "; cin >> y1;
  pelso = Torol(pelso, putolso, x1,y1); // törlés a láncból
  KiirHatra(putolso);
  p = pelso;
  while (p != 0)
  {
    p1 = p->kov; delete p; p = p1;
  }
  pelso = 0; putolso = 0;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Adatok szama: 2

x: 1
y: 1

x: 2
y: 2

A lancolt lista elemei (elore)
    X    Y
    1    1
    2    2

A lancolt lista elemei (elore)
    X    Y
    1    1
    2    2
    5    6
   11    7

Pont torlese
 x: 2
 y: 2

A lancolt lista elemei (visszafele)
    X    Y
   11    7
    5    6
    1    1
```

Hibajelzést kapunk, ha a törlendő pont koordinátája nincs a láncban:

```
Pont torlese
 x: 3
 y: 3

Hibas adat: (3,3) nincs a lancban
```

10. Fájlkezelés struktúrák felhasználásával

Ebben a fejezetben bináris állomány írására és visszaolvasására mutatunk be példákat.

10.1 Struktúrák írása állományba és olvasása állományból

A *Lista* feladatot dolgozzuk át úgy, hogy a *Pont* struktúrák adatait először egy bináris állománya írjuk, majd visszaolvasva listába fűzzük! (*ListaFile*)

A C++ nyelv nem rendelkezik *állománykezelési* utasításokkal, minden feladatot könyvtári függvényekkel, illetve objektumokkal kell megoldanunk.

Bináris állomány írásának menete:

- Definiáljuk az ***ofstream*** osztály példányát:

```
ofstream out;
```

- Az ***open()*** tagfüggvénnyel írásra nyitjuk meg a megadott nevű állományt.

```
out.open(fn);
```

- Ellenőrizzük, hogy sikerült-e a művelet:

```
if (out.fail())
{
   cerr << fn << " letrehozasa nem sikerult\n";
   cin.get();
   return 1;
}
```

- A *p* struktúra kiírása a megnyitott állományba:

```
out.write((char*)&p, sizeof(Pont));
```

- Az állomány lezárása:

```
out.close();
```

Olvasás bináris állományból:

- Definiáljuk az ***ifstream*** osztály példányát:

```
ifstream inp;
```

- Az ***open()*** tagfüggvénnyel olvasásra nyitjuk meg a megadott nevű állományt.

```
inp.open(fn);
```

- Ellenőrizzük, hogy sikerült-e a művelet:

```
if (inp.fail())
{
   cerr << fn << " allomany nem letezik!\n";
   cin.get();
   return 1;
}
```

- A *p* struktúra olvasása a megnyitott állományból:
  ```
  inp.read((char*)&p, sizeof(Pont));
  ```
- Az állomány lezárása:
  ```
  inp.close();
  ```

```
//LISTA.H
struct Pont {
                int x,y;
                Pont *kov, *elozo;
             };

int KiirFile(char fn[]);
int OlvasFile(char fn[], Pont *&e, Pont *&u);
void KiirElore(Pont* e);
void KiirHatra(Pont* u);
Pont* BeszurV(Pont *v, int x1, int y1);
Pont* Torol(Pont *e, Pont *&u, int x1, int y1);

//LISTAFG.CPP
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include "Lista.h"
int KiirFile(char fn[])
{
 Pont p;
 int db;
 ofstream out;
 out.open(fn);
 if (out.fail())
 {
   cerr << fn << " letrehozasa nem sikerult\n";
   cin.get();
   return 1;
 }
 else
 {
   cout << "Adatok kiirasa allomanyba!\n";
   cout << "Adatok szama: ";
   cin >> db;
   for (int i = 0; i< db; i++)
   {
     cout << "\nx: "; cin >> p.x;
     cout << "y: "; cin >> p.y;
     out.write((char*)&p, sizeof(Pont));
   }
   out.close();
   return 0;
 }
}
```

```
int OlvasFile(char fn[], Pont *&e, Pont *&u)
{ ifstream inp;
  Pont p, *uj, *elso = 0, *utolso = 0;
  inp.open(fn);
  if (inp.fail())
  { cerr << fn << " allomany nem letezik!\n"; cin.get();
    return 1;
  }
  else
  {
    cout << "\nAdatok olvasasa allomanybol!\n";
    inp.read((char*)&p, sizeof(Pont));
    while (!inp.eof())
    {
      cout << p.x << " " << p.y << endl;
      uj = new Pont;
      uj->x = p.x; uj->y = p.y;
      uj->kov = 0; uj->elozo = 0;
      if (elso == 0) elso = uj;
      else
      {
        utolso->kov = uj;
        uj->elozo = utolso;
      }
      utolso = uj;
      inp.read((char*)&p, sizeof(Pont));
    }
    inp.close();
    u = utolso; e = elso;
    return 0;
 }
}

// A következő függvények azonosak a Lista feladatban lévőkkel:
void KiirElore(Pont* e)
void KiirHatra(Pont* u)
Pont* BeszurV(Pont* v, int x1, int y1)
Pont* Torol(Pont *elso, Pont *&utolso, int x1, int y1)
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Lista.h"
int main()
{ int x1, y1;
  Pont *pelso, *putolso, *p, *p1;
  char fnev[20];
  cout << "Allomany neve: "; cin >> fnev;
  if (KiirFile(fnev) == 0)
  {
    cin.get();
    if (OlvasFile(fnev, pelso, putolso)==0) // olvas az állományból
    {
      KiirElore(pelso);
      cout << "\nPont torlese\n";
      cout << " x: "; cin >> x1;
      cout << " y: "; cin >> y1;
      pelso = Torol(pelso, putolso, x1,y1); // töröl a láncból
      putolso = BeszurV(putolso,11,7); // beszúr a láncba
      KiirHatra(putolso);
     }
     else
     {
       cerr << "Hiba az olvasaskor!" << endl; cin.get(); return 1;
     }
  }
  else
  {
    cerr << "A kiiras nem sikerult! "<< endl; cin.get(); return 1;
  }
  p = pelso;
  while (p != 0)
  {
    p1 = p->kov; delete p; p = p1;
  }
  pelso = 0; putolso = 0;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Allomany neve: Koord.dat        A lancolt lista elemei (elore)
Adatok kiirasa allomanyba!          X    Y
Adatok szama: 3                     1    1
                                    2    2
x: 1                                3    3
y: 1                            Pont torlese
                                 x: 1
x: 2                             y: 1
y: 2
                                A lancolt lista elemei (visszafele)
x: 3                                X    Y
y: 3                                3    3
                                    2    2
Adatok olvasasa allomanybol!
1 1
2 2
3 3
```

Nem létező adat törlésekor hibajelzést kapunk:

```
Pont torlese                        A lancolt lista elemei (visszafele)
 x: 5                                   X    Y
 y: 5                                   3    3
                                        2    2
Hibas adat: (5,5) nincs a lancban       1    1
```

II. Az objektum-orientált C++ nyelv

11. Bevezetés az objektum-orientált C++ nyelvbe

Az objektum-orientált programozási nyelvek sokkal strukturáltabbak, modulárisabbak és absztraktabbak, mint a hagyományos programozási nyelvek.

Ennek jelentősége az alábbiakban rejlik:

- Az adatok (adattagok) és az adatokat kezelő függvények (tagfüggvények) egységbezárása (*encapsulation*).
- A nagyméretű programok kialakításához szükséges adatrejtés (*data hiding*).
- Az operator *overloading* (operátor átdefiniálás) mechanizmusa.
- Az öröklés (*inheritance*) lehetősége.
- Az új osztály származtatása több alaposztályból (*multiple inheritance*).
- A származtatott osztály örökli az alaposztály(ok) tulajdonságait (adattagjait, tagfüggvényeit), ezek azonban meg is változtathatók:
  - lehetőség van új tagok hozzáadására,
  - a tagfüggvények újradefiniálására (*redefine*),
  - az örökölt tagok elérhetőségének megváltoztatására.
- A fordítás során megvalósított összerendelés a korai kötés (*early binding*).
- Amikor a program futása közben dől el, hogy melyik tagfüggvényt kell aktivizálni, a késői kötésről (*late binding*) beszélünk.
- A virtuális (**virtual**) tagfüggvények teszik lehetővé az objektumok sokalakúságát (*polymorphism*).

A mintafeladatok célja, hogy bemutassák és rávilágítsanak az objektum-orientált C++ nyelv előzőekben felsorolt tulajdonságaira és lehetőségeire.

Néhány feladatot több változatban is bemutatunk, hiszen csak így tanulhatjuk meg a statikus és a dinamikus helyfoglalású objektumok kezelését, a konstruktorok és a destruktor szerepét, és használatát, továbbá a virtuális tagfüggvények jelentőségét stb.

11.1 Mintafeladat: Műveletvégzés struct használatával

Mintafeladatként olyan program működését mutatjuk be, amely a négy alapműveletet hajtja végre. A feladat különféle módon történő megoldása – elméleti kiegészítésekkel – segíti a hagyományos programozásból az objektum-orientált programozásra való áttérést.

Megjegyzés: A mintafeladatokban, amelyek végigvezetik ezt a gondolatmenetet, nem törekszünk a hibás adatok kezelésére. A programban inkább tisztán az elméleti részekre koncentrálunk és a hibakezelésre egy külön fejezetben visszatérünk. A ***main()***

függvény általában szövegesen is magyarázza a kiírt eredményeket – a jobb megértés érdekében.

Írjunk olyan műveletvégző programot, amely függvények használatával végzi el a négy alapműveletet, és külön függvény írja ki a részletes eredményt! A feladat megoldásához használjunk struktúrát! (*MuvStr*)

A *Muvelet* struktúra a művelet működéséhez szükséges adatokat tartalmazza:

- az operandusokat a **double** típusú *x,y* adattagok,
- a műveleti jelet a **char** típusú *muv* adattag és
- a művelet eredményét a **double** *eredmeny* adattag tárolja:

```
struct Muvelet{
                 double x, y, eredmeny;
                 char muv;
               };
```

A *Szamol()* függvény referencia-paraméterként kapja a *Muvelet* típusú struktúrát az adattagok (*x*, *y*, *muv*) alapján elvégzi a műveletet, melynek az eredménye az *eredmeny* adattagba kerül. Megjegyezzük, hogy paraméterként a struktúrát átvéve, leolvashatjuk az adattagokat (*bemenő paraméterek*) a számításhoz, és az eredményt (*kimenő paraméter*) is visszaadhatjuk. A műveleti jel a **switch** utasítás használatával könnyen kiválasztható.

```
void Szamol(Muvelet& sz);
```

A *Kiir()* függvény paraméterében konstans referenciaként kapja meg a *Muvelet* típusú struktúrát – hogy a ne változtathassa meg az adattagokat – és kiírja a művelet eredményét.

```
void Kiir(const Muvelet& sz);
```

A ***main()*** függvény definiál egy *Muvelet* típusú *w* változót. Rákérdez az adatokra, és beolvassa azokat a *w.x*, *w.y* és a *w.muv* adattagokba. A program csak a +,-,*,/ műveleti jeleket fogadja el. Ezután aktiválja a *Szamol()* függvényt a *w* paraméterrel, majd a *Kiir()* függvényt, amely a *w* paraméterben az eredményt is megkapja.

A feladat megoldása:

```
#include <iostream>
#include <cstring> // a strchr() függvény használatához szükséges
using namespace std;
struct Muvelet{
                 double x, y, eredmeny;
                 char muv;
              };
```

```
// A függvények prototípusai
void Szamol(Muvelet& sz);
void Kiir(Muvelet sz);

//MAIN.CPP
int main()
{
  Muvelet w;
  cout << "1.adat: "; cin >> w.x;  // adatok beolvasása
  cout << "2.adat: "; cin >> w.y;
  do              // műveleti jel vizsgálata
  {
    cout << "muveleti jel (+,-,*,/): "; cin >> w.muv;
  }while (!strchr("+-*/",w.muv));
  Szamol(w);
  Kiir(w);
  cin.get(); cin.get();
  return 0;
}
void Szamol(Muvelet& sz)
{
  switch (sz.muv) // a műveleti jel kiválasztása
  { // a kiválasztott műveletnek megfelelő utasítás végrehajtása
    case '+': sz.eredmeny = sz.x + sz.y; break;
    case '-': sz.eredmeny = sz.x - sz.y; break;
    case '*': sz.eredmeny = sz.x * sz.y; break;
    case '/': sz.eredmeny = sz.x / sz.y; break;
  }
}
void Kiir(const Muvelet& sz)
{
  cout << sz.x << " " << sz.muv << " " << sz.y
       << " = " << sz.eredmeny << endl;
}
```

A program futásának eredményei:

```
1.adat: 2.1
2.adat: 3.4
muveleti jel (+,-,*,/): +
2.1 + 3.4 = 5.5

1.adat: 5
2.adat: 6
muveleti jel (+,-,*,/): *
5 * 6 = 30
```

Írjunk olyan műveletvégző programot, amely *MuvStr* programhoz hasonlóan működik, de a feladat megoldásához használjunk egymásba ágyazott struktúrát! (*MuvStr2*)

```
#include <iostream>
#include <cstring>
using namespace std;

struct Adatok{ double a,b; };
struct MuveletiJel{ char muv; };
struct Muvelet {
                 Adatok op;
                 MuveletiJel muvjel;
                 double eredmeny;
               };
void Szamol(Muvelet& sz);
void Kiir(const Muvelet& sz);
int main()
{
  Muvelet w;
  cout << "1. adat: ";  cin >> w.op.a;
  cout << "2. adat: ";  cin >> w.op.b;
  do
  {
    cout << "muveleti jel: "; cin >> w.muvjel.muv;
  }while (!strchr("+-*/",w.muvjel.muv));
  Szamol(w);
  Kiir(w);
  cin.get(); cin.get();
  return 0;
}
void Szamol(Muvelet& sz)
{
 switch (sz.muvjel.muv )
 {
   case '+' : sz.eredmeny = sz.op.a + sz.op.b; break;
   case '-' : sz.eredmeny = sz.op.a - sz.op.b; break;
   case '*' : sz.eredmeny = sz.op.a * sz.op.b; break;
   case '/' : sz.eredmeny = sz.op.a / sz.op.b; break;
 }
}
void Kiir(const Muvelet& sz)
{
  cout << sz.op.a << " " << sz.muvjel.muv << " "
       << sz.op.b << " = " << sz.eredmeny << endl;
}
```

A program futásának eredménye:

```
1. adat: 5
2. adat: 6
muveleti jel (+,-,*,/): *
5 * 6 = 30
```

12. Osztályok

A C++ nyelvben speciális típusok, az osztályok szolgálnak arra, hogy az adatokat és a hozzájuk rendelt függvényeket egy egységként, egy objektumként kezeljük.

A C++ nyelvben a **class** típus a C nyelv struktúratípusának kiterjesztése. Mindkét struktúra tartalmazhat adatmezőket (adattag – *data members*), azonban a C++-ban ezen adattagokhoz különféle műveleteket, ún. tagfüggvényeket (*member fuction*) is megadhatunk. A C++-ban egy osztály típusú tárolási egység függvényeket kombinál adatokkal, és az így létrejött kombinációt elrejtjük, elzárjuk a külvilág elől. Ezt értjük az *egységbezárás* alatt. Egy osztály deklarációja sokban hasonlít a jól ismert struktúra **class** deklarációjához:

```
osztálynév
{
   adattagok
 public:
   tagfüggvények
};
```

A hagyományos C struktúrák (**struct**) és a C++ osztályok (**class**) között a fő különbség a tagokhoz való hozzáférésben keresendő. A C-ben egy struktúra adattagja (megfelelő érvényességi tartományon belül) szabadon elérhetők, míg a C++-ban egy **struct** vagy **class** minden egyes tagjához való hozzáférés önállóan kézben tartható azáltal, hogy nyilvánosnak, privátnak vagy védettnek deklaráljuk a **public**, **private** és **protected** kulcsszavakkal.

A **class** kulcsszóval definiált osztályban az adatmezők alapértelmezés szerint **private** hozzáférésűek, bár ez az elérési kulcsszavakkal módosíthatók.

Az objektum-orientált programozásra jellemző, hogy az adattagokat privátnak (**private**), a tagfüggvényeket pedig publikusnak (**public**) deklaráljuk. Fontos megjegyezni, hogy a C++ osztály önállóan is típusértékű, nem kell a **typedef** segítségével típusazonosítót hozzárendelni.

Összefoglalva: A C++-ban az objektumoknak megfelelő tárolási egység típusát osztálynak (**class**) nevezzük. Az objektum tehát valamely osztálytípussal definiált változó, amelyet más szóhasználattal az osztály példányának (*objektumpéldánynak*) nevezünk (*instance*).

Az alábbi mintafeladatban a már struktúrával megoldott, egyszerű műveletvégző feladatot használjuk fel az objektum-orientált programtervezés és programírás főbb lépéseinek bemutatására. A későbbi feladatok kapcsán az egyszerű, nem objektum-orientált programhoz némileg hasonló megoldástól jutunk el a bonyolultabb, az objektum-orientált programozás lényegét és előnyeit bemutató megoldásokig.

Az objektum-orientált programokhoz tartozó állományoknak a feladatok megoldása során alkalmazott elnevezése és szerepe:

- A ***main()*** függvényt tartalmazó *main.cpp* állomány.
- Az osztály(ok) deklarációját tartalmazó *Osztaly.h* állomány.
- Az osztály(ok) tagfüggvényeinek definícióját tartalmazó *Osztaly.cpp* állomány.
- A program állományait a *Project1.dev* fogja össze.
- A futtatható program neve azonos a projekt nevével, pl. *Project1.exe.*

12.1 Az osztály tagjainak elérése

Tervezzünk olyan objektumot, amely a műveletvégzéshez szükséges **public** hozzáféréssel tárol adatokat, tagfüggvényekkel végzi el a négy alapműveletet, az eredményt pedig a művelet jelölésével írja ki! A feladat megoldása során használjunk statikus helyfoglalású objektumpéldányt! (*MuvPubl*)

Az osztály deklarációját az *Osztaly.h* állományban adjuk meg.

A *Muvelet* osztály nyilvános adattagjai:

double	*x, y*	az operandusok,
double	*eredmeny*	tárolja a művelet eredményét,
char	*muv*	a műveleti jelet tartalmazza.

A *Muvelet* osztály nyilvános tagfüggvényei:

- Az *Init()* tagfüggvény az objektumpéldány adattagjainak ad kezdőértéket:

  ```
  void Init(double x1, double y1, char muv1);
  ```

  A paraméterlistán általában érdemes a paramétereket az adattagok nevéről elnevezni, de a név utolsó karaktereként egy számjeggyel mégis ajánlott megkülönböztetni azoktól.
- A művelet végrehajtását, azaz a számítást a *Szamol()* tagfüggvény végzi:

  ```
  void Szamol();
  ```

  Ennek a **void** típusú függvénynek nincs paraméterlistája, hiszen az objektum adattagjait használja a műveletvégzéshez. Az *x,y* és a *muv* adattagoknak az *Init()* tagfüggvény ad értéket. A *Szamol()* tagfüggvény feladata, hogy a *muv* alapján kiválasztott műveletet végrehajtsa, és az eredményt az *eredmeny* adattagban tárolja.

- Az eredmény közlése a *Kiir()* tagfüggvény feladata:

  ```
  void Kiir();
  ```

A *Kiir()* tagfüggvénynek az adattagokban tárolt adatokat írja ki szöveges formában.

```
//OSZTALY.H
class Muvelet
{
  public:                    // publikus hozzáférés az adattagokhoz
    double x, y, eredmeny;
    char muv;
  public:                    // publikus hozzáférés a tagfüggvényekhez
    void Init(double x1, double y1, char muv1);
    void Szamol();
    void Kiir();
};     // az osztály deklarációját a } után ; vesszővel kell lezárni
```

Az Osztaly.cpp állomány tartalma:

Az osztály tagfüggvényeinek definícióját, azaz a függvények kidolgozását az *Osztaly.cpp* állományban tároljuk. A tagfüggvény neve előtt a **::** hatókör operátorral jelezzük, hogy melyik osztályhoz tartozik. Például, az *Init()* tagfüggvény a *Muvelet* osztályhoz tartozik:

```
void Muvelet::Init(double x1, double y1, char muv1)
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
 void Muvelet::Init(double x1, double y1, char muv1)
 {
   x = x1; y = y1; muv = muv1;
 }
 void Muvelet::Szamol()
 {
  switch (muv)
  {
    case '+': eredmeny = x + y; break;
    case '-': eredmeny = x - y; break;
    case '*': eredmeny = x * y; break;
    case '/': eredmeny = x / y; break;
  }
 }
 void Muvelet::Kiir()
 {
  cout << x << " " << muv << " " << y
       << " = " << eredmeny << endl;
 }
```

A *main.cpp* állomány tartalmazza ***main()*** függvényt. A ***main()*** függvény előtt az *Osztaly.h* fejlécállomány beépítésének mindenképpen szerepelnie kell, hiszen benne található a *Muvelet* osztály deklarációja.

A ***main()*** függvényben sokszor szöveges kiírásokkal magyarázzuk az eredményeket, valamint azt, hogy a feladat milyen elméleti kérdés bemutatására készült.

Mivel a *Muvelet* osztály adattagjai **public** hozzáférésűek, ezért a ***main()*** függvényben hozzáférünk a *w* objektumpéldány *eredmeny* adattagjához, így közvetlenül kiírhatjuk az eredményt:

```
     cout << "Eredmeny: " << w.eredmeny << endl;
```

```
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  Muvelet w;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  { cout << "muveleti jel (+,-,*,/): "; cin >> m;
  }while (!strchr("+-*/",m));
  // az Init() tagfüggvény inicializálja a w objektumpéldány adattagjait
  w.Init(a1,b1,m);
  // a Szamol() tagfüggvény elvégzi a kijelölt műveletet
  w.Szamol();
  // a public adattag közvetlen elérése
  cout << "Eredmeny: " << w.eredmeny << endl;
  // a Kiir() tagfüggvény megjeleníti a részletes eredményt
  w.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 3.2
2.adat: 4.5
muveleti jel (+,-,*,/): +
Eredmeny: 7.7
3.2 + 4.5 = 7.7
```

Módosítsuk a *MuvPubl* programot úgy, hogy a *Muvelet* osztálynak a műveletvégzéshez szükséges adatai **private** hozzáférésűek legyenek! Mit tapasztalunk? (*MuvPriv1*)

```
//OSZTALY.H
class Muvelet
{
  private:    // módosítás privát hozzáférésre
    double x, y, eredmeny;
    char muv;
  public:
    void Init(double x1, double y1, char muv1);
    void Szamol();
    double GetEredmeny(){ return eredmeny; } //inline
    void Kiir();
};

//MAIN.CPP
int main()
{
  ...
  Muvelet w;
  ...
  cout << "Eredmeny: " << w.eredmeny << endl;
  ...
}
```

Azt tapasztaljuk, hogy a ***main()*** függvényben már nem érjük el a *w* objektumpéldány adattagjait, mert a **private** hozzáférés a külvilág elől elzárja azokat. Erről a fordítóprogram hibaüzenettel tájékoztat bennünket.

Módosítsuk az előző programot úgy, hogy a *Muvelet* osztály a műveletvégzéshez ugyan az adattagokat **private** hozzáféréssel tárolja, de az eredmény lekérdezésére egy nyilvános lekérdező tagfüggvényt biztosít! (*MuvPriv2*)

Az *Osztaly.h* állományban a *Muvelet* osztály deklarációját bővítjük a *GetEredmeny()* tagfüggvénnyel, mely az *eredmeny* adattagot függvényértékként adja vissza. A deklaráció állományban a tagfüggvények definícióját csak (implicit) *inline* módon helyezhetjük el. (Az **inline** kulcsszót itt nem kell használnunk!) Az *inline* megadási mód csak akkor javasolt, ha a tagfüggvény viszonylag rövid:

```
    double GetEredmeny(){ return eredmeny; } //inline
```

```
//OSZTALY.H
class Muvelet
{
   private:
    double x, y, eredmeny;
    char muv;
   public:
    void Init(double x1, double y1, char muv1);
    void Szamol();
    double GetEredmeny(){ return eredmeny; } //inline
    void Kiir();
};
```

A ***main()*** függvényben a *w* objektumpéldány eredményt visszaadó függvényének aktiválása:

```
    cout << "Eredmeny: " << w.GetEredmeny() << endl;
```

A program futásának eredménye:

```
1.adat: 4.2
2.adat: 2.0
muveleti jel (+,-,*,/): *
Eredmeny: 8.4
4.2 * 2 = 8.4
```

12.2 Az osztályok friend mechanizmusa

A **friend** (*barát*) mechanizmus lehetővé teszi, hogy az osztály **private** és **protected** tagjait nem saját tagfüggvényből is elérjük. A **friend** deklarációt az osztály deklarációjában belül kell elhelyeznünk, paraméterként pedig az objektumpéldányt kell átadni.

Tervezzünk a *Muvelet* osztálynak egy barát függvényt, amely hatványt számol, azaz az *x* adattagját az *y* adattagra emeli, a hatványozás eredményét pedig függvényértékként adja vissza! (*MuvBarat*)

A barátfüggvény definíciója:

```
friend double Hatvanyoz(const Muvelet& v); // v bemenő paraméter
```

A függvény a paraméterében konstans referenciaként kapja meg a *Muvelet* osztály példányát, hogy a ne változtathassa meg az adattagokat, az eredményt pedig visszatérési értékként adja vissza.

```
//OSZTALY.H
class Muvelet
{
   private:
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(double x1, double y1, char muv1);
    void Szamol();
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
    friend double Hatvanyoz(const Muvelet& v);
};
```

Az *Osztály*.cpp állományban a **friend** függvény kidolgozása is látható. (Megjegyezzük, hogy ekkor már nem kell kitenni a **friend** kulcsszót).

```
#include <iostream>
#include <cmath>
using namespace std;
#include "Osztaly.h"
...
```

```
// friend függvény
double Hatvanyoz(const Muvelet& v)
{
  return pow(v.x,v.y);
}
```

A *main.cpp* állomány a ***main()*** függvényében a *Hatvanyoz* barátfüggvényt hívjuk:

```
. . .
int main()
{
  double a1, b1;
  char m;
  . . .
  Muvelet w(a1,b1,m);
  w.Szamol();
  w.Kiir();
  cout << "Hatvany: " << Hatvanyoz(w) << endl; // barátfüggvény hívása
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 2
2.adat: 3
muveleti jel (+,-,*,/): +
2 + 3 = 5
Hatvany: 8
```

A futtatási eredményben jól látható a barátfüggvény hívásának eredménye, amely az *1.adat* alapot emeli a *2.adat* kitevőre, azaz $2^3 = 8$.

12.3 Statikus helyfoglalású objektumpéldány

A statikus helyfoglalású objektumpéldányokat a program adatterületén hozza létre a fordító. Több példány definiálása esetén csak az adattagokat többszörözi, így minden egyes példány saját adatterülettel rendelkezik. Az objektumok tagfüggvényei azonban csak egyetlen példányban kerülnek a kódterületre, és az objektumpéldányok közösen használják a tagfüggvényeket.

Minden tagfüggvény, még a paraméter nélküliek (**void**) is rendelkeznek egy nem látható (*implicit*) paraméterrel: ez a **this**, amelyben a hívás során az aktuális objektumpéldányra mutató pointert ad át a C++, és minden adattag hivatkozás automatikusan

```
this->adattag
```

kifejezésként kerül a kódba.

Használjuk a főfüggvényben két statikus helyfoglalású *Muvelet* objektumpéldányt! Az egyiket a billentyűzetről beolvasott adatokkal inicializáljuk, a másiknak pedig konstansként adjunk kezdőértéket! (*MuvStat*)

```
//MUVSTAT
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  Muvelet w1, w2;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
    cout << "muveleti jel (+,-,*,/): "; cin >> m;
  }while (!strchr("+-*/",m));
  w1.Init(a1,b1,m);
  w1.Szamol();
  w1.Kiir();
  cout << "\nKonstans parameteru Init(3.2,2.0,'-') hivasa\n";
  w2.Init(3.2,2.0,'-');
  w2.Szamol();
  w2.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 5
2.adat: 6
muveleti jel (+,-,*,/): *
5 * 6 = 30

Konstans parameteru Init(3.2,2.0,'-') hivasa
3.2 - 2 = 1.2
```

13. Konstruktorok és destruktorok

Ebben a fejezetben a konstruktorok és a destruktorok használatát mutatjuk be statikus és dinamikus helyfoglalású objektumpéldányok, illetve dinamikus adattagok esetében.

13.1 Konstruktorok

A C++ programokban az objektumok inicializálását speciális tagfüggvények, a konstruktorok végzik. A konstruktor neve azonos az osztály nevével, azonban nincs visszatérési értéke, még **void** típusú sem. Feladata a dinamikus adattagok létrehozása és az adattagok kezdőértékkel történő ellátása.

A *Muvelet* osztály inicializálására definiáljunk konstruktort! (*MuvKonst*)

A *Muvelet* osztály konstruktora:

```
Muvelet(double x, double y, char muv);
```

A konstruktor neve azonos az osztály nevével. Ha a konstruktor paraméterlistáján a paraméterek neve megegyezik az osztály adattagjainak a nevével, akkor az értékadásnál az adattag előtt használnunk kell a **this** pointert, mert csak ezzel tudjuk megkülönböztetni az adattagot a paramétertől.

```
this->x = x; this->y = y; this->muv = muv;
```

Javaslat az, hogy lehetőleg ne használjunk az adattagokkal azonos paraméterneveket! A konstruktort ebben a feladatban az *Osztaly.h* állományban, *inline* módon dolgoztuk ki.

```
//OSZTALY.H
class Muvelet
{
   private:
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(double x, double y, char muv) // inline
    {
      this->x = x; this->y = y; this->muv = muv;
    }
    void Szamol();
    double GetEredmeny(){ return eredmeny; } //inline
    void Kiir();
};
```

Az *Osztaly.cpp* tartalma nem változott.

```
//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  { cout << "muveleti jel (+,-,*,/): "; cin >> m;
  }while (!strchr("+-*/",m));
  Muvelet w1(a1,b1,m);  // paraméteres konstruktor
  w1.Szamol();
  w1.Kiir();
  cout << "\nKonstans ertekkel hivott Muvelet(3.2,2.0,'-')\n";
  Muvelet w2(3.2,2.0,'-'); // konstanssal hívott konstruktor
  w2.Szamol();
  w2.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 3.2
2.adat: 4.6
muveleti jel (+,-,*,/): +
3.2 + 4.6 = 7.8

Konstans ertekkel hivott Muvelet(3.2,2.0,'-')
3.2 - 2 = 1.2
```

A *Muvelet* osztály inicializálására definiáljunk háromféle konstruktort! (*MuvKonst3*)

Háromféle konstruktort készíthetünk:

- A paraméter nélküli, alapértelmezett (*default*) konstruktor törzsében az adattagoknak konstans kezdőértéket adunk.

  `Muvelet(){ x = 1; y = 1; muv = '+';}`

- Paraméteres konstruktornak az a feladata, hogy azoknak az adattagoknak adjon értéket, amelyek a feladat végrehajtásához szükségesek. Ehhez megfelelő darabszámú és típusú paraméterlistát kell kialakítanunk. Egyébként a konstruktor az eredményt tároló adattagoknak is adhat konstans kezdőértéket, ha szükséges.

  `Muvelet(double x1, double y1, char muv1);`

- Másoló konstruktor (*copy constructor*) használatával egy objektumpéldánynak kezdőértéket adhatunk egy már létező és inicializált másik objektumpéldánnyal.

  `Muvelet(const Muvelet &v);`

Az osztálynak több konstruktora lehet, és mindig az argumentumlista alapján dől el, hogy melyik változatot kell aktivizálni. Ha mi magunk nem definiáltunk konstruktort, akkor a C++ fordító biztosít egy alapértelmezett és egy másoló konstruktort. Ha valamilyen saját konstruktort készítünk, ahhoz az alapértelmezett (paraméter nélküli) konstruktort is definiálnunk kell, amennyiben szükségünk van rá.

```
//OSZTALY.H
class Muvelet
{
   private:
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(){ x = 1; y = 1; muv = '+';} //alapértelmezett
    Muvelet(double x1, double y1, char muv1); // paraméteres
    Muvelet(const Muvelet &v); //másoló konstruktor
    void Szamol();
    void Kiir();
};

//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"

// paraméteres konstruktor
Muvelet::Muvelet(double x1, double y1, char muv1)
{ x = x1; y = y1; muv = muv1; }

// másoló konstruktor (copy construktor)
Muvelet::Muvelet(const Muvelet &v)
{
  x = v.x; y = v.y; muv = v.muv;
}

// void Muvelet::Szamol()  tagfüggvény nem változott
// void Muvelet::Kiir()    tagfüggvény nem változott


//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  Muvelet w1; // alapértelmezett konstruktor hívása
  w1.Szamol();
  w1.Kiir();
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
```

```
   do
   {
     cout << "muveleti jel (+,-,*,/): "; cin >> m;
   }while (!strchr("+-*/",m));

   Muvelet w2(a1,b1,m); // a paraméteres konstruktor hívása
   w2.Szamol();
   w2.Kiir();
   Muvelet w3(w2); // a másoló konstruktor hívása
   w3.Kiir();
   cin.get(); cin.get();
   return 0;
}
```

A program futásának eredménye:

```
1 + 1 = 2
1.adat: 3
2.adat: 5
muveleti jel (+,-,*,/): *
3 * 5 = 15
3 * 5 = 15
```

Amint az eredményekből látható a *másoló konstruktor* a *3* és az *5,* valamint a * adattagokat lemásolta, ezért látunk két azonos eredményt.

13.2 Destruktorok

A konstruktorokban a **new** operátor segítségével az objektum számára lefoglalt memória-terület mindaddig lefoglalt marad, míg fel nem szabadítjuk azt. A C++ nyelvben az ily módon lefoglalt területek felszabadításáról egy speciális tagfüggvényben, az ún. destruktorban gondoskodhatunk.

- A destruktor nevét hullámkarakterrel (~) egybeírt osztálynévként kell megadni, nincs visszatérési típusa, és nincsenek paraméterei sem.
- A **new** operátorral létrehozott objektumok esetén a destruktor aktivizálása csak a **delete** operátorral lehetséges.
- A destruktor virtuális függvény is lehet.
- Ha magunk nem definiálunk destruktort egy adott osztályhoz, a C++ fordító a saját alapértelmezése szerinti változatot használja.

A *Muvelet* osztály tagfüggvényeit bővítsük destruktorral! Használjunk dinamikus helyfoglalású objektumpéldányt! (*MuvDim1*)

A *Muvelet* osztály destruktora:

```
~Muvelet();
```

A destruktor jelenleg üres, mivel az osztály adattagjai nem dinamikusak.

```
//OSZTALY.H
class Muvelet
{
  private:
    double x, y, eredmeny;
    char muv;
  public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    void Szamol();
    void Kiir();
};

//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
Muvelet::Muvelet(double x1, double y1, char muv1)   // konstruktor
{ x = x1; y = y1; muv = muv1; }
Muvelet::~Muvelet()
 { }  // üres dektruktor
void Muvelet::Szamol()
{
  switch (muv)
  {
    case '+': eredmeny = x + y; break;
    case '-': eredmeny = x - y; break;
    case '*': eredmeny = x * y; break;
    case '/': eredmeny = x / y; break;
  }
 }
 void Muvelet::Kiir()
 {
  cout << x << " " << muv << " " << y << " = " << eredmeny << endl;
 }

//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
     cout << "muveleti jel (+,-,*,/): "; cin >> m;
  }while (!strchr("+-*/",m));
  Muvelet *pw; // dinamikus objektumpéldány létrehozása
  pw = new Muvelet(a1,b1,m); // inicializálása konstruktorral
  pw->Szamol();
  pw->Kiir();

  delete pw; // dinamikus objektumpéldány törlése
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 5
2.adat: 6
muveleti jel (+,-,*,/): *
5 * 6 = 30
```

Használjunk két dinamikus objektumpéldányt, melyeket különböző módon inicializálunk! (*MuvDim2*)

```
//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
     cout << "muveleti jel (+,-,*,/): "; cin >> m;
  }while (!strchr("+-*/",m));

  Muvelet *pw1; // dinamikus objektumpéldány létrehozása
  // inicializálása paraméteres konstruktorral
  pw1= new Muvelet(a1,b1,m);
  pw1->Szamol();
  pw1->Kiir();

  // inicializálás konstans értékekkel
  Muvelet *pw2 = new Muvelet(2,3,'*');
  pw2->Szamol();
  pw2->Kiir();

  delete pw1; // dinamikus objektumpéldányok törlése
  delete pw2;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 1.2
2.adat: 4.6
muveleti jel (+,-,*,/): +
1.2 + 4.6 = 5.8
2 * 3 = 6
```

A *Muvelet* osztály adattagjai legyenek dinamikus változók. Módosítsuk a konstruktor és a destruktor tagfüggvényeket! Használjunk dinamikus helyfoglalású objektumpéldányt! (*MuvDimAdat*)

```
//OSZTALY.H
class Muvelet
{
  private:
    double *x, *y, *eredmeny;  // az adattagok mutatók
    char *muv;
  public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    void Szamol();
    void Kiir();
};

//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"

Muvelet::Muvelet(double x1, double y1, char muv1) // konstruktor
{
  x = new double; y = new double; // adattagok létrehozása new
  eredmeny = new double;          // operátorral
  muv = new char;
  *x = x1; *y = y1; *muv = muv1;  // adattagok inicializálása
}
 Muvelet::~Muvelet()     // destruktor
 {
    delete x; delete y; delete eredmeny; // adattagok
    delete muv;                         // memória-területének felszabadítása
}
void Muvelet::Szamol()
{
  switch (*muv)
  {
    case '+': *eredmeny = *x + *y; break;
    case '-': *eredmeny = *x - *y; break;
    case '*': *eredmeny = *x * *y; break;
    case '/': *eredmeny = *x / *y; break;
  }
}
```

```
void Muvelet::Kiir()
 {
  cout << *x << " " << *muv << " " << *y
       << " = " << *eredmeny << endl;
 }

//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"

int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
      cout << "muveleti jel (+,-,*,/): "; cin >> m;
  }while (!strchr("+-*/",m));

  Muvelet *pw; // dinamikus objektumpéldány definiálása
  pw = new Muvelet(a1,b1,m); // inicializálása
  pw->Szamol();
  pw->Kiir();
  delete pw; // dinamikus objektumpéldány felszabadítása

  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 2
2.adat: 7
muveleti jel (+,-,*,/): -
2 - 7 = -5
```

14. Operátorok túlterhelése (operator overloading)

Valójában a szabványos C++ operátorokhoz előre definiált ún. *operátor-függvények* tartoznak. Amikor tehát egy C++-ban definiált műveleti jel értelmezését ki szeretnénk terjeszteni egy saját adattípusra, akkor a kérdéses operátor *operátor-függvényére* adunk meg egy újabb paraméter-szignatúrájú változatot.

14.1 Az osztályok bővítése input/output műveletekkel

A C++ nyelv lehetővé teszi, hogy az osztályokon alapuló adatfolyamoknak (***istream*** - adatbevitel, ***ostream*** - adatkivitel) „megtaníthassuk” saját készítésű osztályunk objektumainak kezelését. A műveletek átdefiniált változatát **friend** operátorként kell elkészíteni.

A *Muvelet* osztály adattagjainak beolvasására, illetve kiírására definiáljuk át az >> és az << operátorokat! (*MuvOpt1*)

```
//OSZTALY.H
#include <iostream>
using namespace std;
class Muvelet
{
   private:
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(){ x = 1; y = 1; muv = '+'; }
    Muvelet(double x1, double y1, char muv1);
    Muvelet(const Muvelet& v);
    void Szamol();
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
    friend istream & operator >> (istream&, Muvelet&);
    friend ostream & operator << (ostream&, const Muvelet&);
};
```

```
/OSZTALY.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"

 Muvelet::Muvelet(double x1, double y1, char muv1)
 {
   x = x1; y = y1; muv = muv1;
 }
```

```
Muvelet::Muvelet(const Muvelet& v)
{
 x = v.x; y = v.y; muv = v.muv;
}
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+': eredmeny = x + y; break;
   case '-': eredmeny = x - y; break;
   case '*': eredmeny = x * y; break;
   case '/': eredmeny = x / y; break;
 }
}
void Muvelet::Kiir()
{
 cout << x << " " << muv << " " << y
      << " = " << eredmeny << endl;
}
// >> adatbevitel átdefiniálása (túlterhelése)
istream & operator >> (istream& is, Muvelet& v)
{
  cout << "1.adat: "; is >> v.x;
  cout << "2.adat: "; is >> v.y;
  do
  {
    cout << "Muveleti jel (+,-,*,/): ";is >> v.muv;
  }while (!strchr("+-*/",v.muv));
}
// << adatkivitel átdefiniálása (túlterhelése)
ostream & operator << (ostream& os, const Muvelet& v)
{
 os << v.x << " " << v.muv << " " << v.y << " = "
    << v.eredmeny << endl;
}
```

Az átdefiniált >> operátor a ***cin*** adatfolyamtól a *w* objektumpéldány adattagjait olvassa be:

```
cin>> w;
```

Az átdefiniált << operátor a ***cout*** adatfolyamra írja a *w* objektumpéldány adattagjait:

```
cout << w;
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{
  Muvelet w;
  cout << "Muvelet osztaly adattagjainak beolvasasa\n";
  cin >> w;
  w.Szamol();
  cout << "Muvelet osztaly adattagjainak kiirasa\n";
  cout << w;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Muvelet osztaly adattagjainak beolvasasa
1.adat: 3.2
2.adat: 6.1
Muveleti jel (+,-,*,/): +
Muvelet osztaly adattagjainak kiirasa
3.2 + 6.1 = 9.3
```

A *Muvelet* osztályok *eredmeny* adattagjainak összeadására a + operátort, az *eredmeny* adattag valós számmal történő szorzására pedig a * operátort definiáljuk át! (*MuvOpt2*)

A *MuvOpt2* a *MuvOpt1* program módosítása, melyben a *Muvelet* osztály tagfüggvénye a + és a * operátorok átdefiniált barátfüggvényeivel bővült:

```
friend Muvelet operator + (Muvelet v1, Muvelet v2);
friend Muvelet operator * (Muvelet v1, double d);
```

A műveletek végzése során az operandusok változatlanok maradnak, és az eredmény egy, a fordító által kezelt ideiglenes objektumban jelenik meg! (A *Muvelet* típusú paramétertől gyorsabb megoldást biztosít a konstans-referencia használata: **const** *Muvelet* &)

```
//OSZTALY.H
#include <iostream>
using namespace std;
class Muvelet
{
   private:
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(){ x = 1; y = 1; muv = '+'; }
    Muvelet(double e){ eredmeny = e; }
    Muvelet(double x1, double y1, char muv1);
    Muvelet(const Muvelet& v);
    void Szamol();
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
    friend istream & operator >> (istream&, Muvelet&);
    friend ostream & operator << (ostream&, const Muvelet&);
    friend Muvelet operator + (Muvelet v1, Muvelet v2);
    friend Muvelet operator * (Muvelet v1, double d);
};
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"

// Az új tagfüggvények
Muvelet operator + (Muvelet v1, Muvelet v2)
{
  return Muvelet(v1.eredmeny + v2.eredmeny);
}
Muvelet operator * (Muvelet v1, double d)
{
  return Muvelet(v1.eredmeny * d);
}
// A többi tagfüggvény nem változott

//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"

int main()
{
  Muvelet w1, w2, w3, w4;
  cout << "w1 osztaly adattagjainak beolvasasa\n";
  cin >> w1;
  w1.Szamol();
  cout << w1;
  cout << "\nw2 osztaly adattagjainak beolvasasa\n";
  cin >> w2;
  w2.Szamol();
  cout << w2;
  w3 = w1 + w2;
  cout<<"\nw1 es w2 eredmenyeinek osszege: "<<w3.GetEredmeny() <<endl;
  w4 = w1 * 3;
  cout <<"\nw1 eredmenyenek haromszorosa: " <<w4.GetEredmeny()<<endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
w1 osztaly adattagjainak beolvasasa
1.adat: 3
2.adat: 5
Muveleti jel (+,-,*,/): +
3 + 5 = 8
w2 osztaly adattagjainak beolvasasa
1.adat: 6
2.adat: 5
Muveleti jel (+,-,*,/): *
6 * 5 = 30

w1 es w2 eredmenyeinek osszege: 38
w1 eredmenyenek haromszorosa: 24
```

15. Az öröklés (öröklődés) mechanizmusa

Az öröklés (*inheritance*) az objektum-orientált C++ nyelv legfőbb sajátossága. Az öröklés azt jelenti, hogy az adott meglévő osztályokból származtatott újabb osztályok öröklik a definiálásukhoz használt alaposztályok már létező adatstruktúráit és függvényeit, ugyanakkor újabb tulajdonságokat is definiálhatnak, vagy régieket újraértelmezhetnek.

Ahhoz, hogy az ős osztályok adattagjai az öröklés folytán az utód osztályban is elérhetők legyenek, **private** elérés helyett **protected** elérést kell használnunk. A **protected** a **private** eléréshez hasonlóan elzárja a tagokat az osztály felhasználója elől, azonban az osztály továbbfejlesztője (örököse) számára elérhetővé teszi őket.

A számatatás szintaxisa a származtatandó osztály neve után kettőspont (**:**), majd a származtatási mód (**public**, **private**, **protected**) az osztály nevével. Többszörös öröklés esetén vesszővel elválasztva újabb osztályokat csatlakoztathatunk a származtatási listához:

class *SzármaztatottOsztály* : **public** *AlapOsztály*

15.1 Származtatott osztályok

A *Muvelet* osztályból származtassunk egy *UjMuvelet* osztályt, amely a *Szamol()* tagfüggvényt újradefiniálja egy olyan változattal, amely a négy alapművelet mellett a hatványozás jelét (^) is kezeli! (*MuvOrok1*)

A *Muvelet* osztály adattagjainak a hozzáférését változtassuk meg **protected**-re, mivel így az örökös látja, de a külvilág felé rejtett marad. Az *UjMuvelet* osztály a *Muvelet* osztálytól **public** módon örököl, és nincs újabb adattagja. A konstruktorának paraméterei megegyeznek a *Muvelet* osztály konstruktorának a paramétereivel, hiszen az adattagokat neki is inicializálni kell, azonban van saját destruktora.

A *Szamol()* tagfüggénye a *Muvelet* osztály *Szamol* tagfüggvényét újradefiniálja, mivel bevezeti a hatványozás műveletét (^), amely az *x* adattagot emeli az *y*, adattagra és az eredményt az *eredmeny* adattagban tárolja.

```
class UjMuvelet : public Muvelet
{
   public:
     UjMuvelet(double x1, double y1, char muv1);
    ~UjMuvelet();
    void Szamol(); // Elfedi az örökölt Szamol() tagfüggvényt
};
```

```
//OSZTALY.H
class Muvelet
{
   protected: //öröklés esetén szerencsésebb a protected elérés használata
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    void Szamol();
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
};

class UjMuvelet : public Muvelet
{
   public:
     UjMuvelet(double x1, double y1, char muv1);
    ~UjMuvelet();
    void Szamol();
};

//OSZTALY.CPP
#include <iostream>
#include <cmath>
using namespace std;
#include "Osztaly.h"

Muvelet::Muvelet(double x1, double y1, char muv1)
{
  x = x1; y = y1; muv = muv1; eredmeny = 0;
}
Muvelet::~Muvelet() {}
void Muvelet::Szamol()
 {
  switch (muv)
   {
    case '+': eredmeny = x + y; break;
    case '-': eredmeny = x - y; break;
    case '*': eredmeny = x * y; break;
    case '/': eredmeny = x / y; break;
   }
 }
 void Muvelet::Kiir()
 {
  cout << x << " " << muv << " " << y << " = " << eredmeny << endl;
 }

// Az UjMuvelet osztály konstruktora
UjMuvelet::UjMuvelet(double x1, double y1, char muv1):
                     Muvelet(x1,y1,muv1) // paraméterek átadva
                                         // a Muvelet konstruktornak
{} // üres a törzse, mivel nincs saját inicializálható adattagja
```

```
// Az UjMuvelet osztály destruktora

UjMuvelet::~UjMuvelet(){}

// Az UjMuvelet osztály Szamol() tagfüggvénye

void UjMuvelet::Szamol()
{
   if (muv == '^')
     eredmeny = pow(x,y);     // ez a bővített művelet
    // a pow() függvény igényli a cmath fejlécállományt
   else
     Muvelet::Szamol();
}

//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"

int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
    cout << "muveleti jel (+,-,*,/,^): "; cin >> m;
  }while (!strchr("+-*/^",m));
  UjMuvelet w1(a1,b1,m);
  w1.Szamol();
  w1.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
1.adat: 2                             1.adat: 2
2.adat: 3                             2.adat: 3
muveleti jel (+,-,*,/,^): *           muveleti jel (+,-,*,/,^): ^
2 * 3 = 6                             2 ^ 3 = 8
```

Az új osztály kezeli mind a négy alapműveletet, és az új, a ^ karakterrel azonosított hatványozás műveletet is.

15.2 Korai kötés - bezáródás jelensége

Legyen a *Muvelet* osztály új tagfüggvénye a *Vegrehajt()*, amely a saját *Szamol()* tagfüggvényét aktiválja! Mint láttuk a *Muvelet* osztálytól származtatott *UjMuvelet* osztálynak is van egy *Szamol()* tagfüggvénye, amely a négy alapművelet mellett kezeli a hatványozás műveletét (^) is! A *UjMuvelet* objektumpéldánya aktiválja az örökölt *Vegrehajt()* tagfüggvényt! Mit tapasztalunk? (*MuvBezar*)

```
//OSZTALY.H
class Muvelet
{
   protected:
    double x, y, eredmeny;
    char muv;
   public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    void Szamol();
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
    void Vegrehajt(){ Szamol(); }  // új inline tagfüggvény
};
class UjMuvelet : public Muvelet
{
   public:
     UjMuvelet(double x1, double y1, char muv1);
    ~UjMuvelet();
    void Szamol();
};
```

Az *Osztaly.cpp* állományban a *Muvelet* osztály *Szamol()* tagfüggvényét módosítottuk: ha a 4 alapművelettől eltérő műveleti jelet kap, akkor hibajelzést ad és az *eredmeny* adattag értékét nullára állítja.

```
void Muvelet::Szamol()
{
  switch (muv)
  {
    case '+': eredmeny = x + y; break;
    case '-': eredmeny = x - y; break;
    case '*': eredmeny = x * y; break;
    case '/': eredmeny = x / y; break;
    default:
       eredmeny = 0;
       cerr << "Ervenytelen muveleti jel: " << muv <<endl;
  }
}
```

```
//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"

int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {  cout << "muveleti jel (+,-,*,/,^): "; cin >> m;
  }while (!strchr("+-*/^",m));
  UjMuvelet w1(a1,b1,m); // w1 UjMuvelet objektumpéldánya
  w1.Vegrehajt();        // aktiválja a Vegrehajt() tagfüggvényt
  w1.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
1.adat: 2                          1.adat: 2
2.adat: 3                          2.adat: 3
muveleti jel (+,-,*,/,^): +        muveleti jel (+,-,*,/,^): ^
2 + 3 = 5                          Ervenytelen muveleti jel: ^
                                   2 ^ 3 = 0
```

Azt tapasztaljuk, hogy a ^ műveleti jelet érvénytelennek jelzi a program, mivel a *Vegrehajt()* a *Muvelet* osztály *Szamol()* tagfüggvényét hajtotta végre. Ez a jelenség a *korai kötés (early bindig)*, mert már a fordítási fázisban eldől, hogy az alaposztály *Szamol* nevű tagfüggvénye hívódik meg.

15.3 Késői kötés - virtuális tagfüggvények

Az előző feladatban keletkezett problémát a virtuális tagfüggvények használatával orvosolhatjuk. Az alaposztály virtuális tagfüggvénye teljesen újradefiniálható (lecserélhető) a származtatott osztály azonos nevű és paraméterezésű tagfüggvényével. Ezzel a származtatási hierarchia valamely szintjén, mindig az éppen ott definiált függvényváltozat lesz érvényben. Így megvalósul az objektumok sokalakúsága (a *polymorphism*).

Az előző *MuvBezar* feladatban annyi módosítást végezzünk el, hogy a *Muvelet* osztályban a *Szamol()* tagfüggvényt tegyük virtuálissá! Az *UjMuvelet* változatlanul aktiválja a *Vegrehajt()* tagfüggvényt! Mit tapasztalunk? (*MuvVirt*)

A feladatot dinamikus objektumpéldány használatával is bemutatjuk (*MuvVirtD*).

```
//OSZTALY.H
class Muvelet
{
  protected:
    double x, y, eredmeny;
    char muv;
  public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    virtual void Szamol();  // virtuális a Szamol() tagfüggvény
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
    void Vegrehajt(){ Szamol(); }
};
class UjMuvelet : public Muvelet
{
   public:
     UjMuvelet(double x1, double y1, char muv1);
    ~UjMuvelet();
    void Szamol();
};

//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
    cout << "muveleti jel (+,-,*,/,^): "; cin >> m;
  }while (!strchr("+-*/^",m));
  UjMuvelet w1(a1,b1,m);
  w1.Vegrehajt();
  w1.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 2
2.adat: 3
muveleti jel (+,-,*,/,^): ^
2 ^ 3 = 8
```

Azt tapasztaljuk, hogy jó lesz az eredmény, amely azt jelenti, hogy a *Vegrehajt()* tagfüggvényben az *UjMuvelet* osztály *Szamol()* tagfüggvénye hívódott meg. Ez az ún. késői kötés, amikor a futás során derül ki, hogy melyik tagfüggvényt kell aktiválni. A

virtual teszi lehetővé a megfelelő tagfüggvények hívását, azaz a késői kötés (*late binding*) megvalósítását.

15.4 Öröklés több osztálytól

Oldjuk meg a mintafeladatot úgy, hogy az adattagok tárolására hozzunk létre önálló osztályokat, és a *Muvelet* osztályt azoktól származtassuk! Az *UjMuvelet* osztály a *Muvelet* osztálytól származtatva változatlanul úgy működjön, mint az előző feladatban láttuk! (*MuvOrok2*)

Három osztályra osztottuk szét az adattagokat:

1. *Adatok* osztály: adattagjai az *x* és *y* operandusok. A konstruktor az adattagoknak ad kezdőértéket.
2. *MuveldetiJel* osztály: adattagja *muv* műveleti jelet tárolja a művelethez. Konstruktora a műveleti jelnek ad kezdőértéket.
3. *Eredmeny* osztály: adattagja a művelet eredményét tároló *eredmeny*. Konstruktora zérus kezdőértéket az *eredmeny* adattagnak.

A *Muvelet* osztály három osztálytól származik:

```
class Muvelet : public Adatok,
                public MuveletiJel,
                public Eredmeny
{
    . . .
};
```

Konstruktora a szükséges kezdőértékadás miatt sorra hívja az alaposztályok konstruktorait, a törzse üres, hiszen nincsen adattagja.

```
Muvelet::Muvelet(double x1, double y1, char muv1):
          Adatok(x1,y1),
          MuveletiJel(muv1),
          Eredmeny()
{ }
```

A teljes program, a megjegyzésben megadott állományokban található:

```
//OSZTALY.H
class Adatok
{
  protected:
    double x,y;
  public:
    Adatok(double x1, double y1) { x = x1; y = y1; }
    ~Adatok(){}
};
```

```
class MuveletiJel
{
  protected:
    char muv;
 public:
    MuveletiJel( char muv1){ muv = muv1;  }
    ~MuveletiJel() {}
};
class Eredmeny
{
   protected:
     double eredmeny;
   public:
     Eredmeny() { eredmeny = 0;}
     ~Eredmeny(){}
};
class Muvelet  : public Adatok,
                 public MuveletiJel,
                 public Eredmeny
{   public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    virtual void Szamol();
    void Vegrehajt() {Szamol(); }
    double GetEredmeny(){ return eredmeny; }
    void Kiir();
};
class UjMuvelet  : public Muvelet
{
   public:
     UjMuvelet(double x1, double y1, char muv1);
    ~UjMuvelet();
    void Szamol();
};

//OSZTALY.CPP
#include <iostream>
#include <cmath>
using namespace std;
#include "Osztaly.h"
Muvelet::Muvelet(double x1, double y1, char muv1):
         Adatok(x1,y1),
         MuveletiJel(muv1),
         Eredmeny()
{ }

Muvelet::~Muvelet() {}

void Muvelet::Szamol()
{
  switch (muv)
  {
    case '+': eredmeny = x + y; break;
    case '-': eredmeny = x - y; break;
    case '*': eredmeny = x * y; break;
    case '/': eredmeny = x / y; break;
    default:
        cerr << "Hibas muveleti jel: " << muv << endl;
  }
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y
       << " = " << eredmeny << endl;
}

UjMuvelet::UjMuvelet(double x1, double y1, char muv1):
             Muvelet(x1,y1,muv1) {}

UjMuvelet::~UjMuvelet(){}

void UjMuvelet::Szamol()
{
   if (muv == '^')
     eredmeny = pow(x,y);
   else
     Muvelet::Szamol();
}

//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"

int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
    cout << "muveleti jel (+,-,*,/,^): "; cin >> m;
  }while (!strchr("+-*/^",m));
  UjMuvelet w1(a1,b1,m);
  w1.Vegrehajt();
  w1.Kiir();

  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.adat: 2
2.adat: 3
muveleti jel (+,-,*,/): ^
2 ^ 3 = 8
```

15.5 Öröklés helyett beágyazás

Az öröklés helyett az osztályok példányait adattagokként is megadhatjuk, ez az ún. beágyazás módszere. El lehet dönteni, hogy melyik alkalmasabb egy feladat megoldására.

A több osztálytól való öröklés feladatát módosítsuk, hogy a *Muvelet* osztály adattagjai az osztályok példányai legyenek! Az *UjMuvelet* osztály a *Muvelet* osztálytól származtatva változatlanul a hatványozást végezze el! (*MuvOrok3*)

```
//OSZTALY.H
class Adatok
{
  public:
    double x,y;
  public:
    Adatok(double x1, double y1)
    { x = x1; y = y1; }
    ~Adatok(){}
};
class MuveletiJel
{
  public:
    char muv;
  public:
    MuveletiJel( char muv1)
    { muv = muv1;  }
    ~MuveletiJel()  {}
};
class Eredmeny
{
   public:
     double eredmeny;
   public:
     Eredmeny() { eredmeny = 0;}
     ~Eredmeny(){}
};
class Muvelet
{
   protected:
    Adatok a;
    MuveletiJel m;
    Eredmeny e;
   public:
    Muvelet(double x1, double y1, char muv1);
    ~Muvelet();
    virtual void Szamol();
    void Vegrehajt() {Szamol(); }
    double GetEredmeny(){ return e.eredmeny; }
    void Kiir();
};
```

```
class UjMuvelet : public Muvelet
{
   public:
     UjMuvelet(double x1, double y1, char muv1);
    ~UjMuvelet();
    void Szamol();
};

//OSZTALY.CPP
#include <iostream>
#include <cmath>
using namespace std;
#include "Osztaly.h"
Muvelet::Muvelet(double x1, double y1, char muv1):
         a(x1, y1),
         m(muv1),
         e()
{ }
Muvelet::~Muvelet(){}
void Muvelet::Szamol()
{
  switch (m.muv)
  {
    case '+': e.eredmeny = a.x + a.y; break;
    case '-': e.eredmeny = a.x - a.y; break;
    case '*': e.eredmeny = a.x * a.y; break;
    case '/': e.eredmeny = a.x / a.y; break;
    default:
        cerr << "Hibas muveleti jel: " << m.muv << endl;
  }
}
void Muvelet::Kiir()
{
  cout << a.x << " " << m.muv << " " << a.y
       << " = " << e.eredmeny << endl;
}
UjMuvelet::UjMuvelet(double x1, double y1, char muv1):
             Muvelet(x1,y1,muv1) {}
UjMuvelet::~UjMuvelet(){}
void UjMuvelet::Szamol()
{
   if (m.muv == '^')
    e.eredmeny = pow(a.x,a.y);
  else
    Muvelet::Szamol();
}

//MAIN.CPP
#include <iostream>
#include <cstring>
using namespace std;
#include "Osztaly.h"
```

```
int main()
{
  double a1, b1;
  char m;
  cout << "1.adat: "; cin >> a1;
  cout << "2.adat: "; cin >> b1;
  do
  {
    cout << "muveleti jel (+,-,*,/,^): "; cin >> m;
  }while (!strchr("+-*/^",m));
  UjMuvelet w1(a1,b1,m);
  w1.Vegrehajt();
  w1.Kiir();

  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
1.adat: 2
2.adat: 3
muveleti jel (+,-,*,/,^): +
2 + 3 = 5

1.adat: 2
2.adat: 3
muveleti jel (+,-,*,/,^): ^
2 ^ 3 = 8
```

Az eredmény teljesen azonos a több osztálytól való öröklés esetén kapottal. Hasonlítsuk össze a megoldásokat, és nézzük meg, hogy melyik az egyszerűbb, és melyik tűnik bonyolultabbnak.

16. Általánosított osztályok (templates)

A sablonosztály általános alakja, ahol a *típus1*, ...*típusn* az általánosított típusokat jelölik:

```
  template <class típus, . . . class típusn> class osztalynev {
  }
```

16.1 Specializáció és példányosítás

A sablonokat különböző módon használhatjuk.

Készítsünk sablont a *Muvelet* osztályra azért, hogy a műveleteket **int**, **float** és **double** típusú operandusokkal is elvégezhessük! (*MuvSablon*)

Az általánosított osztály definícióját deklarációs állományban (.H) ajánlott elhelyezni. A fájl kezdődik az osztályok definíciójával, majd a tagfüggvények kidolgozásával folytatódik. A megfelelő **template <class** *típus*> paramétereket az osztályok és a tagfüggvények elé kell beszúrni, valamint a paraméterek típusát is jelölni kell:

```
template <class R>
class Muvelet
{
   private:
    R x, y;
   . . .
}
template <class R>
Muvelet<R>::Muvelet(R x1, R y1, char muv1)
```

A *Muvelet* osztály sablonjának kidolgozása:

```
//OSZTALY.H
template <class R>
class Muvelet
{
   private:
    R x, y;
    double eredmeny; // az osztás miatt valós típusú
    char muv;
   public:
    Muvelet(R x1, R y1, char muv1);
    ~Muvelet();
    void Szamol();
    void Kiir();
};
```

```
template <class R>
Muvelet<R>::Muvelet(R x1, R y1, char muv1)
{ x = x1; y = y1; muv = muv1; }
template <class R>
Muvelet<R>::~Muvelet()
{ }  // üres dektruktor
template <class R>
void Muvelet<R>::Szamol()
{
  switch (muv)
  {
    case '+': eredmeny = x + y; break;
    case '-': eredmeny = x - y; break;
    case '*': eredmeny = x * y; break;
    case '/': eredmeny = (double) x / y; break; //osztás miatt double
  }
}
template <class R>
void Muvelet<R>::Kiir()
{
  cout << x << " " << muv << " " << y << " = " << eredmeny << endl;
}
```

A *Muvelet* sablon használatát mutatjuk be a *main()* függvényben, ahol **double**, **float** és **int** típusú paraméterrel is aktiváljuk a konstruktor első két paraméterét, amely éppen a két operandus. A művelet eredménye mindenképpen **double** típusú lesz az osztás művelet következtében:

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{
  cout << "double parameter\n";
  Muvelet <double>*p1;
  p1 = new Muvelet<double>(3.56, 2.1, '-');
  p1->Szamol();
  p1->Kiir();
  cout << "\nfloat parameter\n";
  Muvelet <float>*p2;
  p2 = new Muvelet<float>(10.7, 5.2, '+');
  p2->Szamol();
  p2->Kiir();
  cout << "\nint parameter\n";
  Muvelet <int>w3(5, 2, '/');
  w3.Szamol();
  w3.Kiir();
  delete p1;
  delete p2;

  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
double parameter
3.56 - 2.1 = 1.46

float parameter
10.7 + 5.2 = 15.9

int parameter
5 / 2 = 2.5
```

16.2 Sablonok alkalmazása öröklött objektumok esetén

Készítsünk sablont a többszintű öröklésű osztályok számára! Mutassuk be a sablon használatát! (*OrokSablon*)

A *Negyzet* osztályból származtassuk a *Teglalap* osztályt, használjuk fel a négyzet *a* oldalát a téglalap egyik oldalának, a téglalap *b* oldalát pedig definiáljuk a *Teglalap* osztály adattagjaként. A terület és a kerület számításánál a *Teglalap* osztály átdefiniálja a *Negyzet* osztály *Szamol()* tagfüggvényét.

```
//OSZTALY.H
template <class R>
class Negyzet
{
  protected:
    R a; // négyzet oldala
    R ered1, ered2; // ered1 - kerület, ered2 - terület
  public:
    Negyzet(R a1);
    ~Negyzet();
    virtual void Szamol();
    R GetEred1();
    R GetEred2();
};
template <class R>
Negyzet<R>::Negyzet(R a1)
{ a = a1; }
template <class R>
Negyzet<R>::~Negyzet() {};
template <class R>
void Negyzet<R>::Szamol()// a négyzet kerületét és területét számítja
{ ered1 = a*a; ered2 = 4*a;}
template <class R>
R Negyzet<R>::GetEred1()
{ return ered1; }
template <class R>
R Negyzet<R>::GetEred2()
{ return ered2; }
template <class R>
```

```
class Teglalap : public Negyzet<R>
{
  protected:
    R b;   // a teglalap b oldala
  public:
    Teglalap(R a1, R b1);
    ~Teglalap();
  void Szamol();
};
template <class R>
Teglalap<R>::Teglalap(R a1, R b1): Negyzet<R>(a1){ b = b1;}
template <class R>
Teglalap<R>::~Teglalap(){};
template <class R>
void Teglalap<R>::Szamol() //a téglalap kerületét és területét számítja
{ ered1 = a*b; ered2 = 2*(a+b);}

//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{
  Negyzet <int>n1(1);
  n1.Szamol();
  cout << "Negyzet  terulete: " << n1.GetEred1() <<endl;
  cout << "Negyzet  kerulete: " << n1.GetEred2() <<endl;
  Negyzet <double>n2(1.5);
  n2.Szamol();
  cout << "Negyzet  terulete: " << n2.GetEred1() <<endl;
  cout << "Negyzet  kerulete: " << n2.GetEred2() <<endl;
  Teglalap <int>t1(2,3);
  t1.Szamol();
  cout << "\nTeglalap  terulete: " << t1.GetEred1() <<endl;
  cout << "Teglalap  terulete: " << t1.GetEred2() <<endl;
  Teglalap <double>t2(2.5,3.2);
  t2.Szamol();
  cout << "Teglalap  terulete: " << t2.GetEred1() <<endl;
  cout << "Teglalap  terulete: " << t2.GetEred2() <<endl;

  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Negyzet  terulete: 1
Negyzet  kerulete: 4
Negyzet  terulete: 2.25
Negyzet  kerulete: 6

Teglalap  terulete: 6
Teglalap  terulete: 10
Teglalap  terulete: 8
Teglalap  terulete: 11.4
```

17. Objektum-orientált láncolt-lista

Láncolt listák készítésével foglalkozunk ebben a fejezetben. Bemutatjuk az új listaelem láncba – előre, hátra illetve, névsor szerinti – fűzését, a listaelem törlését a láncból és a láncolt listák tartalmának kiíratását (lásd a 9. fejezet).

17.1 Lista készítése előre-hátra láncolással

Írjunk programot, amelyben a *Tanulo* típusú objektumok a tanulók adatait tartják nyilván! A *Tanulo* típusú objektumokat fűzzük láncba a *Lista* osztály segítségével! Számítsuk ki a tanulók átlagát, és keressük meg a maximális ösztöndíjat! (*Tan_01*)

A *Tanulo* osztály tartalmazza egy tanuló adatait: vezetéknév (*vnev*), keresztnév (*knev*), átlag (*atlag*) és ösztöndíj (*osztondij*). A lánc elemeinek összefűzéséhez szükséges a következő (*kov*), illetve az előző (*elozo*) *Tanulo* típusú mutató.

Az osztály konstruktora a tanuló adattagjainak ad kezdőértéket.

```
//OSZTALY.H
class Tanulo
{
  public:
    char vnev[20];
    char knev[20];
    double atlag;
    int osztondij;
    Tanulo *kov;
    Tanulo *elozo;
  public:
    Tanulo(char vnev0[], char knev0[],double atlag0, int osztondij0);
    ~Tanulo();
};
```

A *Lista* osztály csak egy példányban létezik a programban. Ez az osztály tartja nyilván a lánc első (*elso*) és az utolsó (*utolso*) elemére mutató pointert, valamint a *MaxOsztondij* adattagja az évfolyam maximális ösztöndíját és a *TanulokAtlaga* adattagja pedig az évfolyam tanulóinak átlagát.
A *Lista* osztály tagfüggvényei:

- *AddTanulo()* - átveszi egy tanuló adatait és egy *irany* paramétert, mely értékétől függően a lánc elejére (1), vagy a lánc végére (nem 1) fűzi az új *Tanulo* elemet.
- *KiirElore()* - megjeleníti a lista adatait, a láncon előre (*kov*) haladva.
- *Szamol()* - megkeresi a maximális ösztöndíjat, és kiszámítja a tanulók átlagát.
- *GetAtlag()* - visszaadja a *TanulokAtlaga* adattagban kiszámított eredményt.
- *GetMaxOsztondij()* - visszaadja *MaxOsztondij* tartalmát, amely a tanulók maximális ösztöndíját tartalmazza.

```
class Lista
{
  private:
    Tanulo *elso;
    Tanulo *utolso;
    int MaxOsztondij;
    double TanulokAtlaga;
  public:
    Lista();
    ~Lista();
    void AddTanulo(char vnev0[], char knev0[],double atlag0,
                   int osztondij0, int irany);
    void KiirElore();
    void Szamol();
    double GetAtlag(){ return TanulokAtlaga;}
    int GetMaxOsztondij() {return MaxOsztondij;}
};
//OSZTALY.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
// A Tanulo osztály konstruktora
Tanulo::Tanulo(char vnev0[], char knev0[],
               double atlag0, int osztondij0)
{ strcpy(vnev, vnev0);strcpy(knev, knev0);
  atlag = atlag0; osztondij = osztondij0;
  kov = 0; elozo = 0;
}
// A Tanulo osztály desktruktora üres
Tanulo::~Tanulo()
{  }
// Lista osztály konstruktora
Lista::Lista()
{
  elso = 0; utolso = 0;
  TanulokAtlaga = 0; MaxOsztondij = 0;
}
// Lista osztály destruktora
Lista::~Lista()  //destruktor törli a lánc minden elemét a memóriából
{
  Tanulo *p, *p1;
  p = elso;

  while (p != 0)
  {
    p1 = p->kov;
    delete p;  // a lefoglalt láncelem törlése
    p = p1;
  }
  elso = 0; utolso = 0;
}
```

```
// A lánc elejére vagy végére fűzi egy tanuló adatát az irany szerint
void Lista::AddTanulo(char vnev0[], char knev0[],
                      double atlag0, int osztondij0, int irany)
{
  Tanulo *p;
  if (elso == 0) // ez lesz az első
  {
    elso = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    utolso = elso;
  }
  else if (irany == 1)  // első elé fűz
  {
    p = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    p->kov = elso;
    elso->elozo = p;
    elso = p;
  }
  else  // az utolsó mögé fűz
  {
    p = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    utolso->kov = p;
    p->elozo = utolso;
    utolso = p;
  }
}
// A legnagyobb összegű ösztöndíj keresése.
// A tanulók átlagának számítása.
void Lista::Szamol()
{ Tanulo *p;
  int db = 0;
  p = elso;
  while (p!= 0)  // a lánc vége (0) állítja le a ciklust
  { TanulokAtlaga += p->atlag;  // átlagok összegének számítása
    db++;
    if (MaxOsztondij < p->osztondij) // keresi a maximális ösztöndíjat
       MaxOsztondij = p->osztondij;
    p = p->kov;   // a láncon előre halad
  }
  TanulokAtlaga /= db; // tanulók átlagának számítása
}
// A tanulók adatainak megjelenítése
void Lista::KiirElore()
{ Tanulo *p;
  p = elso;
  if (p != 0) cout << endl << "Tanulok adatai" << endl;
  {
    while (p != 0) // a lánc vége állítja le a ciklust
    { cout << "Nev         : " << p->vnev
                             << "  " << p->knev << endl;
      cout << "  atlag     : " << p->atlag << endl;
      cout << "  osztondij: " << p->osztondij << endl;
      p = p->kov;  // a lánc következő elemére lép
    }
  }
}
```

```
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
int main()
{
  int n,osztondij0;
  Lista *p;
  char vnev0[20], knev0[20];
  double atlag0;
  p = new Lista(); // létrehozzuk a Lista osztály objektumpéldányát
  cout << "Tanulok szama: "; cin >> n; //n elemmel feltöltjük a láncot
  if (n > 0)
  {
    for (int i = 0; i< n ; i++)
    {                                   // beolvassuk a tanulók adatait
      cout << "Vezeteknev: ";  cin >> vnev0;
      cout << "Keresztnev: ";  cin >> knev0;
      cout << "Atlag     : ";  cin >> atlag0;
      cout << "Osztondij : ";  cin >> osztondij0;
      cout << endl;
      // tanulók adatait a lánc végére fűzi
      p->AddTanulo(vnev0,knev0,atlag0,osztondij0,2);
    }
    p->KiirElore();
    // kikeresi a maximális ösztöndíjat és kiszámítja az évfolyam átlagát
    p->Szamol();
    cout << endl;
    cout << "Az evfolyam atlaga : " << p->GetAtlag() << endl;
    cout << "Maximalis osztondij: " << p->GetMaxOsztondij()<< endl;
   }
   else
     cout << "Adathiba!" << endl;
   delete p; // a Lista osztály objektumpéldányának törlése
   cin.get(); cin.get();
   return 0;
}
```

A program futásának eredménye:

```
Tanulok szama: 3
Vezeteknev: Kiss
Keresztnev: Laszlo
Atlag     : 4
Osztondij : 4000

Vezeteknev: Nagy
Keresztnev: Endre
Atlag     : 3
Osztondij : 2000

Vezeteknev: Komaromi
Keresztnev: Anna
Atlag     : 5
Osztondij : 8000
Tanulok adatai
Nev        : Kiss  Laszlo
  atlag    : 4
  osztondij: 4000
Nev        : Nagy  Endre
  atlag    : 3
  osztondij: 2000
Nev        : Komaromi  Anna
  atlag    : 5
  osztondij: 8000

Az evfolyam atlaga : 4
Maximalis osztondij: 8000
```

17.2 Listaelem törlése a láncból

Módosítsuk a *Tan_01* feladatot úgy, hogy legyen lehetőség a lánc elemeinek törlésére! A *Lista* osztályt bővítsük egy *letszam* adattaggal, valamint a *GetTanulo()* és a *DelTanulo()* tagfüggvényekkel! (*Tan_02*)

```
//OSZTALY.H
class Tanulo
{
  public:
    char vnev[20];
    char knev[20];
    double atlag;
    int osztondij;
    Tanulo *kov;
    Tanulo *elozo;
  public:
    Tanulo(char vnev0[], char knev0[],
           double atlag0, int osztondij0);
    ~Tanulo();
};
// A Lista osztály tartja nyilván a lánc első és az utolsó elemére
// mutató pointert.
class Lista
{
  private:
    Tanulo *elso;
    Tanulo *utolso;
    int MaxOsztondij;
    double TanulokAtlaga;
    int letszam;    // tanulók létszámának nyilvántartása
  public:
    Lista();
    ~Lista();
    void AddTanulo(char vnev0[], char knev0[],double atlag0,
                   int osztondij0, int irany);
    void KiirElore();  // Az adatok megjelenítése
    void Szamol();
    double GetAtlag(){ return TanulokAtlaga;}
    int GetMaxOsztondij() {return MaxOsztondij;}
    // a lánc egy elemének törlése
    void DelTanulo(char vnev0[], char knev0[]);
    int GetLetszam(){return letszam; }    // létszám lekérdezése
};
//OSZTALY.CPP
#include <iostream>
#include <cstring>
#include "Osztaly.h"
using namespace std;
```

```
// A Tanulo osztály konstruktora
Tanulo::Tanulo(char vnev0[], char knev0[],
               double atlag0, int osztondij0)
{ strcpy(vnev, vnev0); strcpy(knev, knev0);
  atlag = atlag0; osztondij = osztondij0;
  kov = 0; elozo = 0;
}
// A Tanulo osztály destruktora
Tanulo::~Tanulo()
{  }
// A Lista osztály konstruktora
Lista::Lista()
{ elso = 0; utolso = 0;
  TanulokAtlaga = 0; MaxOsztondij = 0;
  letszam = 0;  // létszám alapállapotba állítása
}
// A Lista osztály destruktora törli a lánc elemeit
Lista::~Lista()
{ Tanulo *p, *p1;
  p = elso;
  while (p != 0)
  {
    p1 = p->kov;
    delete p; p = p1;
  }
  elso = 0; utolso = 0;
}
// Egy tanuló hozzáfűzése a lánchoz az irany szerint
void Lista::AddTanulo(char vnev0[], char knev0[],
                      double atlag0, int osztondij0, int irany)
{ Tanulo *p;
  letszam++;      // létszám növelése
  if (elso == 0) // a lánc első eleme
  {
    elso = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    utolso = elso;
  }
  else if (irany == 1)  // a lánc első elem elé fűz
  {
    p = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    p->kov = elso; elso->elozo = p; elso = p;
  }
  else  // az utolsó elem mögé fűz
  {
    p = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    utolso->kov = p; p->elozo = utolso; utolso = p;
  }
}
// Kiszámítja a tanulók átlagát és megkeresi a legnagyobb ösztöndíjat
void Lista::Szamol()
{ Tanulo *p;
  int db = 0;
  p = elso;
  while ( p!= 0)
  { TanulokAtlaga += p->atlag;
```

```
    db++;
    if (MaxOsztondij < p->osztondij) MaxOsztondij = p->osztondij;
    p = p->kov;
  }
  TanulokAtlaga /= db;
}
// Kiírja a tanulók adatait a láncon előre haladva
void Lista::KiirElore()
{ Tanulo *p;
  p = elso;
  if (p != 0) cout << endl << "Tanulok adatai" << endl;
  {
    while (p != 0)
    {
      cout << "Nev         : " << p->vnev
                             << "  " << p->knev << endl;
      cout << "  atlag     : " << p->atlag << endl;
      cout << "  osztondij: " << p->osztondij << endl;
      p = p->kov;
    }
  }
}
```

A tanuló törlése a láncból a vezeték és a keresztnevének megadásával lehetséges. Ha megtaláltuk a törlendő tanulót, akkor meg kell vizsgálni, hogy hol helyezkedik el a láncban (a lánc első, utolsó vagy közbülső eleme) vagy például ő az egyedüli elem-e.

```
void Lista::DelTanulo(char vnev0[], char knev0[]) // tanuló törlése
{ Tanulo *p, *p1, *p2;
  p = elso;
  while (p != 0)
  { // a törlendő tanuló nevének összehasonlítása
    // a láncelemekben tárolt nevekkel
    if (strcmp(p->vnev, vnev0)== 0 && strcmp(p->knev, knev0)== 0 )
    {
      letszam--;    // törlésnél a létszám csökken
      p1 = p->elozo;
      p2 = p->kov;
      if (p1 == 0)
      {
        elso = p2;
        if (p2 != 0)
        { p2->elozo = 0; }
         else
        { utolso = p2; }
      }
      else
      {
        p1->kov = p2;
        if (p2 != 0)
        { p2->elozo = p1; }
        else
        { utolso = p1; }
      }
      p->kov = 0; p->elozo = 0;
      delete p; // a láncból eltávolított elem megsemmisítése
      break;
    }
    p = p->kov;
  }
}

//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;

int main()
{ int n,osztondij0;
  char ch;
  Lista *p;
  char vnev0[20], knev0[20];
  double atlag0;
  p = new Lista();
  cout << "Tanulok szama: "; cin >> n;
  if (n > 0)
  {
   for (int i = 0; i< n ; i++)
   { cout << "Vezeteknev: ";  cin >> vnev0;
     cout << "Keresztnev: ";  cin >> knev0;
     cout << "Atlag     : ";  cin >> atlag0;
     cout << "Osztondij : ";  cin >> osztondij0;
     cout << endl;
     p->AddTanulo(vnev0,knev0,atlag0,osztondij0,2);
   }
   p->KiirElore();
   if (p->GetLetszam())
   { cout << endl << "Adat torlese (i,I): "; cin >> ch;
     // csak i vagy I karakter leütésekor kér törlendő adatot
     // egyéb karakter leütésekor nincs lehetőség a törlésre
     if (ch == 'i' || ch == 'I')
     { cout << "Torlendo vezeteknev: "; cin >> vnev0;
       cout << "         keresztnev: "; cin >> knev0;
       // a nevével azonosított tanuló törlése
       p->DelTanulo(vnev0, knev0); p->KiirElore();
     }
   }
   if (p->GetLetszam()) // ha van még tanuló a listában, újraszámol
   { p->Szamol();
     cout << endl;
     cout << "Az evfolyam atlaga : " << p->GetAtlag() << endl;
     cout << "Maximalis osztondij: " << p->GetMaxOsztondij()<< endl;
     cout << "Letszam            : " << p->GetLetszam() << endl;
   }
  }
  // a Lista osztály objektumpéldányának törlése
  delete p;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Tanulok szama: 4
Vezeteknev: Kiss
Keresztnev: Laszlo
Atlag     : 4
Osztondij : 4000

Vezeteknev: Nagy
Keresztnev: Endre
Atlag     : 3
Osztondij : 2000

Vezeteknev: Komaromi
Keresztnev: Anna
Atlag     : 5
Osztondij : 8000

Vezeteknev: Lazar
Keresztnev: Timea
Atlag     : 2
Osztondij : 1000

Tanulok adatai
Nev        : Kiss  Laszlo
  atlag    : 4
  osztondij: 4000
Nev        : Nagy  Endre
  atlag    : 3
  osztondij: 2000
Nev        : Komaromi  Anna
  atlag    : 5
  osztondij: 8000
Nev        : Lazar  Timea
  atlag    : 2
  osztondij: 1000

Adat torlese (i,I): i
Torlendo vezeteknev: Nagy
         keresztnev: Endre

Tanulok adatai
Nev        : Kiss  Laszlo
  atlag    : 4
  osztondij: 4000
Nev        : Komaromi  Anna
  atlag    : 5
  osztondij: 8000
Nev        : Lazar  Timea
  atlag    : 2
  osztondij: 1000

Az evfolyam atlaga : 3.66667
Maximalis osztondij: 8000
Letszam            : 3
```

17.3 Listaelem beszúrása a láncba névsor szerint

Módosítsuk a *Tan_02* feladatot úgy, hogy az *Lista* osztály *AddNevsor()* tagfüggvénye a tanulókat névsor szerint fűzze be a láncba! (*Tan_03*)

```
//OSZTALY.H
class Lista
{
  private:
    Tanulo *elso;
    Tanulo *utolso;
    int MaxOsztondij;
    double TanulokAtlaga;
    int letszam;
  public:
    Lista();
    ~Lista();
    void AddNevsor(char vnev0[], char knev0[],
                   double atlag0, int osztondij0);
    void KiirElore();
    void Szamol();
    double GetAtlag(){ return TanulokAtlaga;}
    int GetMaxOsztondij() {return MaxOsztondij;}
    void DelTanulo(char vnev0[], char knev0[]);
    int GetLetszam(){return letszam; }
};

//OSZTALY.CPP
#include <iostream>
#include <cstring>
#include "Osztaly.h"
using namespace std;
// Az AddNevsor tagfüggvény a tanulókat névsor szerint fűzi a láncba
void Lista::AddNevsor(char vnev0[], char knev0[],
                      double atlag0, int osztondij0)
{
  Tanulo *p, *p1, *t;
  char nev[25], nev0[45];
  int volt = 0;
  letszam++;         // létszám növelése
  t = new Tanulo(vnev0, knev0, atlag0, osztondij0);
  if (elso == 0) // ez lesz az első
  {
    elso = t; utolso = t; volt = 1;
  }
  else
  {
    p = elso;
    while (p != 0)
    {
      strcpy(nev0, vnev0); strcat(nev0, knev0);
      strcpy(nev,p->vnev); strcat(nev, p->knev);
      if (strcmp(nev0, nev) <0)
      {
        p1 = p->elozo;
        if (p1 != 0)
        { p1->kov =t; t->elozo = p1; t->kov = p; p->elozo =t; volt= 1;}
        else
        { t->kov = elso; elso->elozo = t; elso = t; volt = 1; }
      }
      if (volt == 1) break;
      p = p->kov;
    }
    if (volt == 0)
    {
      utolso->kov = t;
      t->elozo = utolso; utolso = t;
    }
  }
}
// A többi tagfüggvény nem változott
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;

int main()
{
  int n,osztondij0;
  char ch;
  Lista *p;
  char vnev0[20], knev0[20];
  double atlag0;
  p = new Lista();
  cout << "Tanulok szama: "; cin >> n;
  if (n > 0)
  {
   for (int i = 0; i< n ; i++)
   { cout << "Vezeteknev: ";  cin >> vnev0;
     cout << "Keresztnev: ";  cin >> knev0;
     cout << "Atlag      : ";  cin >> atlag0;
     cout << "Osztondij : ";  cin >> osztondij0;
     cout << endl;
     p->AddNevsor(vnev0,knev0,atlag0,osztondij0);
   }
   p->KiirElore();
   if (p->GetLetszam())
   { cout << endl << "Adat torlese (i,I): "; cin >> ch;
     if (ch == 'i' || ch == 'I')
     { cout << "Torlendo vezeteknev: "; cin >> vnev0;
       cout << "         keresztnev: "; cin >> knev0;
       p->DelTanulo(vnev0, knev0);
       p->KiirElore();
     }
   }
   if (p->GetLetszam())
   { p->Szamol(); cout << endl;
```

```
     cout << "Az evfolyam atlaga : " << p->GetAtlag() << endl;
     cout << "Maximalis osztondij: " << p->GetMaxOsztondij()<< endl;
     cout << "Letszam            : " << p->GetLetszam() << endl;
   }
  }
  delete p;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Tanulok szama: 4
Vezeteknev: Kiss
Keresztnev: Laszlo
Atlag     : 4
Osztondij : 4000

Vezeteknev: Nagy
Keresztnev: Endre
Atlag     : 3
Osztondij : 2000

Vezeteknev: Komarom
Keresztnev: Anna
Atlag     : 5
Osztondij : 8000

Vezeteknev: Lazar
Keresztnev: Timea
Atlag     : 2
Osztondij : 1000

Tanulok adatai
Nev        : Kiss  Laszlo
  atlag    : 4
  osztondij: 4000
Nev        : Komaromi  Anna
  atlag    : 5
  osztondij: 8000
Nev        : Lazar  Timea
  atlag    : 2
  osztondij: 1000
Nev        : Nagy  Endre
  atlag    : 3
  osztondij: 2000
```

```
Adat torlese (i,I): i
Torlendo vezeteknev: Kiss
         keresztnev: Laszlo

Tanulok adatai
Nev        : Komaromi  Anna
  atlag    : 5
  osztondij: 8000
Nev        : Lazar  Timea
  atlag    : 2
  osztondij: 1000
Nev        : Nagy  Endre

atlag    : 3
osztondij: 2000

Az evfolyam atlaga : 3.33333
Maximalis osztondij: 8000
Letszam            : 3
```

18. Numerikus módszerek objektum-orientált felfogásban

Ebben a fejezetben néhány olyan algoritmust és numerikus módszert mutatunk be, amelyek osztályt használnak a műveleteik végrehajtásához.

18.1 Interpoláció

Írjunk programot, amelyben az *Interpolacio* osztály alkalmas lineáris és *Langrange* típusú interpoláció végzésére! (lásd a 8.3 fejezetet) (*Interpolacio*)

```
//OSZTALY.H
class InterPolacio
{
  private:
    double *x, *y;
    int n;
    double xh,yh;
  public:
    InterPolacio();
    ~InterPolacio();
    int Linint();
    int Lagrange();
    double GetYh(){ return yh;}
    double GetXh(){ return xh;}
    void SetXh(double xx) { xh = xx; }
    void Olvas();
    void Rendez();
    void Kiir();
};

//OSZTALY.CPP
#include <iostream>
#include <iomanip>
#include "Osztaly.h"
using namespace std;
// konstruktor
InterPolacio::InterPolacio()
{
   do
   {
     cout << "\nElemek szama (min. 2): "; cin >> n;
   } while (n < 2);
   x = new double[n]; y = new double[n];
   for (int i = 0; i<n; i++)
   {
     cout << endl << "x[" <<i << "] = "; cin >> x[i];
     cout << "y[" <<i << "] = "; cin >> y[i];
   }
   cout << endl << "Iterpolalando ertek (xh): "; cin >> xh;
}
```

```
InterPolacio::~InterPolacio()//destruktor
{
  delete[] x;  delete[] y;
}
int InterPolacio::Linint() // lineáris interpoláció
{ int i;
  yh = 0;
  if (xh < x[0] || xh > x[n-1])
  {
     cerr << endl << "Hibajelzes:" << endl;
     cerr << "Hibas az interpolalando ertek: " << xh << endl;
     cerr << "Kiesik " << x[0] <<" - "<< x[n-1] <<" hatarbol!"<< endl;
     return 1;
  }
  else
  {
   for (i = 0; i<n-1; i++)
   { if (x[i+1] == x[i])
       {
         cerr << endl << "Hibajelzes:" << endl;
         cerr << "Azonosak az x koordinatak: "<< endl;
         cerr << i << " .elem : " << x[i] << endl;
         cerr << i+1 << " .elem : " << x[i+1] << endl;
         return 1;
       }
    if (x[i] <= xh && x[i+1] >= xh)
    { yh = y[i] +(y[i+1]-y[i])/(x[i+1]-x[i])*(xh-x[i]);
      return 0;
    }
   }
  }
}
//Lagrange interpoláció
int InterPolacio::Lagrange()
{ double p = 0.0, s;
  for (int i = 0; i< n; i++)
  { s = 1.0;
    for (int j = 0; j<n; j++)
    { if (i != j)
      { if (x[i] == x[j])
        {
          cerr << endl << "Hibajelzes:" << endl;
          cerr << "Azonosak az x koordinatak: "<< endl;
          cerr << i << " .elem : " << x[i] << endl;
          cerr << i+1 << " .elem : " << x[i+1] << endl;
          return 1;
        }
        s *=(xh-x[j])/(x[i] - x[j]);
      }
    }
    p += y[i]*s;
  }
  yh= p;
  return 0;
}
```

```
//A n darabszámú x- és y-koordináták újraolvasása
void InterPolacio::Olvas()
{  for (int i = 0; i< n; i++)
   {
      cout << endl << "x[" <<i << "] = "; cin >> x[i];
      cout << "y[" <<i << "] = "; cin >> y[i];
   }
   // interpolálandó érték beolvasása
  cout << endl << "Iterpolalando ertek (xh): "; cin >> xh;
}
//rendezés az x koordináták növekvő sorrendje szerint
void InterPolacio::Rendez()
{
  double sx, sy;
  for (int i = 0; i< n-1; i++)
  {
    for (int j = i+1; j<n; j++)
    {
      if (x[j] < x[i])
      {
        sx = x[i];
        x[i] = x[j];
        x[j] = sx;
        sy = y[i];
        y[i] = y[j];
        y[j] = sy;
      }
    }
  }
}
// a rendezett adatok megjelenítése
void InterPolacio::Kiir()
{ cout << endl << "Rendezett adatok" << endl;
  cout << setw(10) << "x" << setw(10) << "y" << endl;
  for (int i = 0; i<n; i++)
  {
    cout << setw(10) << x[i] << setw(10) << y[i] << endl;
  }
  cout << endl;
}
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
int main()
{ InterPolacio p1;
  p 1.Rendez(); p1.Kiir();
  if (p1.Linint() == 0)
  {
     cout << "Linearis interpolacio\n";
     cout << "Interpolacio helye(xh): " << p1.GetXh() << endl;
     cout << "Intepolalt ertek (yh) : " << p1.GetYh() << endl << endl;
  }
  InterPolacio *p2;
  p2 = new InterPolacio;
```

```
   p2->Rendez(); p2->Kiir();
   if (p2->Lagrange() == 0)
   {
      cout << " Lagrange interpolacio\n";
      cout << "Interpolacio helye(xh): " << p2->GetXh() << endl;
      cout << "Intepolalt ertek (yh) : " << p2->GetYh() << endl;
   }
   delete p2;
   cin.get(); cin.get();
   return 0;
}
```

A program futásának eredménye:

```
Elemek szama (min. 2): 4

x[0] = 1
y[0] = 1

x[1] = 2
y[1] = 1.8

x[2] = 3
y[2] = 2.2

x[3] = 4
y[3] = 2.5

Iterpolalando ertek (xh): 2.5

Rendezett adatok
         x       y
         1       1
         2     1.8
         3     2.2
         4     2.5

Linearis interpolacio
Interpolacio helye(xh): 2.5
Intepolalt ertek (yh) : 2
```

```
Elemek szama (min. 2): 4

x[0] = 1
y[0] = 1

x[1] = 3
y[1] = 2.2
b
x[2] = 2
y[2] = 1.8

x[3] = 4
y[3] = 2.5

Iterpolalando ertek (xh): 2.5

Rendezett adatok
         x       y
         1       1
         2     1.8
         3     2.2
         4     2.5

Lagrange interpolacio
Interpolacio helye(xh): 2.5
Intepolalt ertek (yh) : 2.03125
```

18.2 Integrálás

Írjunk programot, amelyben néhány integrálási módszert mutat be az *Integral* osztály definiálásával. A tagfüggvények a különféle integrálási módszerek alapján függvényre mutató pointer felhasználásával működjenek! Számítsuk ki π értékét a különféle integrálási módszerek alkalmazásával! (*Integral*)

```
//INTEGRAL.H
typedef double f(double x);
typedef f *fp;
```

```
class Integral
{
  protected:
     double alsoh, felsoh;
     int osztas;
     double Eredmeny;
     fp fg;
  public:
    Integral(double a, doule b, int n, fp f1)
    { alsoh  = a;
      felsoh = b;
      if (n % 2 != 0 ) n++;
      osztas = n;
      fg = f1;
      Eredmeny = 0;
    }
    void Teglalap();
    void KisTrapez();
    void NagyTrapez();
    void Erinto();
    void KisSimpson();
    void NagySimpson();
    double GetEredmeny() { return Eredmeny; }
};
//INTEGRAL.CPP
#include <iostream>
using namespace std;
#include "Integral.h"
// Integrál számítása téglalap módszerrel
void Integral::Teglalap()
{
  double s = 0, h = (felsoh-alsoh)/osztas;
  Eredmeny = 0;
  for (int i = 1; i<osztas; i++)
  {
    Eredmeny += (*fg)(alsoh+h*i);
  }
  Eredmeny = h*((*fg)(alsoh) + Eredmeny + (*fg)(felsoh));
}

// Integrál számítása kis trapéz módszerrel
void Integral::KisTrapez()
{
  Eredmeny = (felsoh-alsoh)*((*fg)(alsoh) + (*fg)(felsoh))/2.;
}
// Integrál számítása nagy trapéz módszerrel
void Integral::NagyTrapez()
{ double  s = 0, h;
  h = (felsoh - alsoh)/osztas;
  for (int i = 1; i< osztas; i++)
  {
    s +=(*fg)(alsoh + h*i);
  }
  Eredmeny = h*((*fg)(alsoh)/2 + s + (*fg)(felsoh)/2);
}
```

```
void Integral::Erinto() // Integrál számítása érintő módszerrel
{
  double h;
  h = (felsoh - alsoh)/osztas;
  Eredmeny = 0;
  for (int i = 1; i<= osztas; i++)
  {
    Eredmeny += (*fg)(alsoh + (i-0.5)*h);
  }
  Eredmeny *= h;
}
// Integrál számítása kis Simpson módszerrel
void Integral::KisSimpson()
{
  Eredmeny = (felsoh - alsoh)/6.*((*fg)(alsoh)
             + 4.*(*fg)((alsoh + felsoh)/2) + (*fg)(felsoh));
}
// Integrál számítása nagy Simpson módszerrel
void Integral::NagySimpson()
{
  double s = 0, h;
  h = (felsoh - alsoh)/osztas;
  for (int i = 1; i< osztas; i++)
  {
    s +=  4.*(*fg)(alsoh + i*h)*(i % 2)
        + 2.*(*fg)(alsoh+i*h)*((i+1)%2);
  }
  s += (*fg)(alsoh) + (*fg)(felsoh);
  Eredmeny = s * h/3.0;
}
//MAIN.CPP
#include <iostream>
#include <iomanip>
using namespace std;
#include "Integral.h"
double fg1(double x);
int main()
{
  double alsoH, felsoH, IntegralErteke;
  int felosztas;
  cout.precision(14);  // az értékes jegyek számának beállítása
  cout << "Integral szamitasa " << endl << endl;
  cout << "Also hatar : "; cin >> alsoH;
  cout << "Felso hatar: "; cin >> felsoH;
  cout << "Intervallum felosztasa: "; cin >> felosztas;
  Integral I(alsoH, felsoH, felosztas, fg1);
  I.Teglalap();
  cout <<endl<< "Teglalap modszerrel     : " << I.GetEredmeny()<< endl;
  I.KisTrapez();
  cout <<endl<< "Kis Trapez modszerrel   : " << I.GetEredmeny()<< endl;
  I.NagyTrapez();
  cout <<endl<< "Nagy Trapez modszerrel : " << I.GetEredmeny()<< endl;
  I.Erinto();
  cout <<endl<< "Erinto modszerrel       : " << I.GetEredmeny()<< endl;
  I.KisSimpson();
  cout <<endl<< "Kis Simpson modszerrel : " << I.GetEredmeny()<< endl;
  I.NagySimpson();
  cout <<endl<< "Nagy Simpson modszerrel: " << I.GetEredmeny()<< endl;
  cin.get(); cin.get();
  return 0;
}
// π értékét számítjuk ki integrállal
// alsó határ: 0 felső határ: 1 osztások száma: 2000
double fg1(double x)
{ return 4.0/(1. +x*x); }
```

A π konstans értéke 20 jegyre kerekítve: 3.141 592 653 589 793 238 5

A program futásának eredménye:

```
Integral szamitasa

Also hatar : 0
Felso hatar: 1
Intervallum felosztasa: 2000

Teglalap modszerrel     : 3.1430926119231

Kis Trapez modszerrel   : 3

Nagy Trapez modszerrel : 3.1415926119231

Erinto modszerrel       : 3.1415926744231

Kis Simpson modszerrel : 3.1333333333333

Nagy Simpson modszerrel: 3.1415926535898
Integral szamitasa
```

Az eredmények magukért beszélnek.

18.3 Gyök meghatározása érintő módszerrel

Írjunk programot, amelyben nemlineáris egyenlet gyökét érintő (*Newton-Raphson*) módszerrel határozzuk meg! (*Nemlin*)

```
//OSZTALY.H
typedef double f(double x);
typedef f *fp;
class NewtonR
{ protected:
    double x0;
    double eps;
    double gyok;
    fp f1;
    fp df1;
  public:
    NewtonR(double x01, double eps1, fp fg, fp dfg);
    ~NewtonR(){};
    int NewtonRaphson();
    double GetGyok() { return gyok; }
};
```

```
//OSZTALY.CPP
#include <iostream>
#include <cmath>
using namespace std;
#include "Osztaly.h"

NewtonR::NewtonR(double x01, double eps1,fp fg,fp dfg)
{ x0 = x01; eps = eps1; f1 = fg; df1 = dfg; }

int NewtonR::NewtonRaphson()
{ double xe, nevezo;
  xe = x0;
  do
  { nevezo = df1(xe);  // azonos (*df1)(xe)
    if (fabs(nevezo) <1.0e-6)
    { cerr << "Nulla a nevezo!"; cin.get(); return 0; }
    xe -= f1(xe)/nevezo;
  }while (eps < fabs( f1(xe)));
  gyok = xe;
  return 1;
}
//MAIN.CPP
#include <iostream>
#include <cmath>
using namespace std;
#include "Osztaly.h"
double fx(double x);
double dfx(double x);
int main()
{ double x02, eps2;
  cout << "Gyok meghatarozasa erinto modszerrel\n";
  cout << "(Newton+Raphson modszer)\n";
  cout << "Intevallum eleje: "; cin >> x02;
  cout << "Hibakorlat (1.0e-6): "; cin >> eps2;
  NewtonR  nr(x02,eps2,fx,dfx);
  if (nr.NewtonRaphson())
  { cout << "\nGyok: " << nr.GetGyok() << endl; }
  cin.get(); cin.get();
  return 0;
}

double fx(double x)
{ return pow(x,3)+x-0.7; }

double dfx(double x)
{ return 3*pow(x,2)+1;}
```

A program futásának eredménye:

```
Gyok meghatarozasa erinto modszerrel
(Newton+Raphson modszer)
Intevallum eleje: 0
Hibakorlat (1.0e-6): 1e-6

Gyok: 0.541351
```

19. Egy osztállyal feljebb léphetünk?

Könyvünk előző 18 fejezetében a C++ nyelv alapjaival ismerkedhetett meg az Olvasó. Annak érdekében, hogy a lényegre lehessen koncentrálni, egyszerű, jól érthető mintapéldákból építkeztünk. Ebben a részben néhány olyan C++ alkalmazást készítünk, melyek megértése ugyan több erőfeszítést igényel, azonban a jutalom nem marad el, hiszen a C++ nyelv felhasználásának távlatai nyílnak meg előttünk.

19.1 A C++ nyelv bővítése a halmaztípussal

Tekintsük át, hogy milyen C++ eszközök szükségesek a halmaztípus létrehozásához! (*Halmazelo*)

Először definiáljuk a *Halmaz* osztályt, melyet később ellátunk halmazalgebrai műveletekkel. A létrehozandó *Halmaz* típus, az egyszerűség kedvéért legyen az egész számok halmaza. Az ehhez szükséges minimális osztálydefiníció a következő:

```
// OSZTALY.H
class halmaz
{
        int  *elemek;                        // az elemek tömbje
        int  darab;                          // a halmaz számossága
public:
         Halmaz(int n=0, int *e=NULL);       // konstruktor
         ~Halmaz(){if (darab) delete [] elemek;} // destruktor
};
```

A *Halmaz* osztály objektumát a halmaz számosságának és elemeinek ismeretében hozzuk létre. Ennek megfelelően a konstruktor az *OSZTALY.CPP* állományban:

```
Halmaz::Halmaz(int n, int e[])
{
  int i;

  if (n)
    for (i = 0, darab = n, elemek = new int[n]; i < n; i++)
       elemek[i] = e[i];
  else
  {
    darab = 0;
    elemek = NULL;
  }
}
```

A halmaz aktuális elemeit az *elemek* mutatóval megcímzett dinamikusan létrehozott egész tömbben tároljuk, a halmaz számosságát pedig a *darab* tagban tartjuk nyilván.

Mint láttuk, a konstruktor paraméterként egy egész számot és egy egész tömböt vár, feladata pedig az elemek tárolására szolgáló tömb létrehozása és feltöltése az argumentumként átadott tömb elemeivel. Ha a konstruktort paraméter nélkül aktivizáljuk, akkor üres halmaz jön létre, így az *elemek* mutató értéke *NULL* lesz. A destruktor nagyon egyszerű: a nem üres halmaz elemeit tároló, dinamikusan foglalt memória-területet felszabadítja, míg üres halmaz esetén nincs teendője. Mivel nagyon egyszerű a destruktorfüggvény, az *inline* megvalósítást választottuk. Az osztálydefiníció alapján a következőképpen definiálhatunk halmazobjektumokat (*MAIN.CPP*):

```
int szamok[] = {1,2,3,4,5}; // ilyen elemekkel töltjük fel a halmazt

Halmaz a(5,szamok);           // elemei: 1,2,3,4,5
Halmaz b;                     // üres halmaz
```

Az *a* halmaz 5 egész számot fog tartalmazni, melyeket a *szamok* tömbben adunk meg, a *b* halmaz pedig üres halmaz lesz. Eddigi definíciónk alapján fenti halmazaink nem módosíthatók, például a *b* halmaz mindig üres halmaz marad. Nyilvánvaló, hogy az egyik legfontosabb művelet, amelynek értelmezését az imént definiált *Halmaz* típusra ki kell terjesztenünk, az értékadás művelete. Az alábbiakban megadunk egy olyan operátorfüggvényt, amely az = operátort (az értékadás műveletét) kiterjeszti a *Halmaz* típusra:

```
Halmaz& Halmaz::operator=(const Halmaz &s)
{
  int i;
  if (darab) delete [] elemek;       // a korábbi elemek[] eldobása
  darab = s.darab;
  if (!s.darab) { elemek = NULL; }
  else           // új halmaz létrehozása a jobbérték halmazból
  {
    elemek = new int[darab];
    for (i = 0; i < darab; i++)
       elemek[i] = s.elemek[i];
  }
 return *this;
}
```

Így minden olyan esetben, amikor egy halmazt egy másik halmaznak értékül adunk, ez az operátorfüggvény hívódik meg, tehát a fenti *b* halmaz is kaphat értéket ennek segítségével:

```
b = a;  // a Halmaz::operator=()függvény aktiválása
```

Valamely osztály operátorfüggvényei is lehetnek többértelműek, feltéve, hogy paraméter-szignatúrájuk egyedi. Például, a *Halmaz* típus esetén értelmezhetünk egy másik értékadást is, amellyel egyetlen elemet tartalmazó halmazok hozhatók létre:

```
b=12;
```

A fenti művelet elvégzéséhez szükséges átdefiniált függvényváltozat:

```
Halmaz& Halmaz::operator=(const int i)
{
  if (darab) delete [] elemek;
  darab = 1;
  elemek = new int[darab];
  elemek[0] = i;
  return *this;
}
```

Ahhoz, hogy eljussunk a halmazalgebrai műveletekhez, az operátor-tagfüggvények mellett barát (**friend**) operátorfüggvényeket is kell használnunk. Vannak esetek, amikor a C++ nyelv mindkét megoldást támogatja, ekkor további megfontolások alapján kell eldöntenünk, hogy melyiket is válasszuk, hiszen ugyanazt a műveletet csak egyféleképpen definiálhatjuk. Nézzünk is mindjárt egy példát, melyben a halmazok unióját (egyesítését) az összeadás operátorához (kétoperandusú +) rendeljük:

```
class Halmaz {
              friend Halmaz &operator+(Halmaz &a, Halmaz &b);
              ...
             };
```

vagy

```
class halmaz {
             ...
             public:
               Halmaz &operator+(Halmaz &a);
             ...
             }
```

Figyeljük meg, hogy amikor tagfüggvényként definiáltuk az összeadás műveletét, akkor eggyel kevesebb paramétert adtunk meg explicit módon. Ennek az a magyarázata, hogy egy tagfüggvény első operandusa mindig az az implicit osztályobjektum, amelyik az operátorfüggvényt aktivizálta. Például, az értékadás művelete esetén a

```
b = a;
```

utasítás ekvivalens a

```
b.operator=(a);
```

függvényhívással, ahol az operátorfüggvényen belül a **this** pointer *b*-re fog mutatni. Az összeadás műveleténél a tagfüggvényként történő megvalósítás esetleg hátrányos is lehet, mert például a **this** mutatóval megcímzett első operandust elronthatjuk, ami nyilván nem kívánt mellékhatás.

Vannak olyan operátorok, amelyekhez tartozó operátorfüggvény csak tagfüggvényként definiálható. Ezek az operátorok az = (értékadás), az `[]` (indexelő operátor), az

`()` (aktivizáló operátor) és a `->` (pointeres tagkiválasztó operátor). Egy tagfüggvényként definiált operátor bal oldali operandusa mindig a saját osztályába kell tartozzon. Ha egy operátor bal oldali operandusa idegen típusú, akkor a kérdéses operátorfüggvény csak **friend** függvény lehet. Így például egy halmaz elemeinek a szokásos `<<` operátorral való kiírása a ***cout*** szabványos kimeneti adatfolyamba a következőképpen oldható meg:

```
#include <iostream>
class Halmaz {  ...
                friend ostream& operator<<(ostream &os, Halmaz &s);
                ...
              };
...
ostream& operator<<(ostream &os, Halmaz &s)
{ int i;
  if (!s.darab)                    // Üres halmaz
  {
    os << "Üres.";
    return os;
  }
  // Az elemeket ciklusban írjuk ki:
  for (i = 0; i < s.darab; os << s.elemek[i++] << ' ') ;
  return os;
}
```

Az output művelettel kibővített *Halmaz* osztály használatát az alábbi főfüggvény szemlélteti:

```
#include "Osztaly.h"

int main()
{
  int szamok[] = {1,2,3,4,5}; // Ilyen elemekkel töltjük fel a halmazt
  Halmaz a(5,szamok);         // Elemei: 1,2,3,4,5
  cout<<a<<endl;
  Halmaz b;                   // Üres halmaz
  cout<<b<<endl;
  b=12;
  cout<<b<<endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1 2 3 4 5
ures halmaz
12
```

Készítsük el az egész elemek tárolására használható *Halmaz* osztályt, a szokásos halmazalgebrai műveletekkel! (*Halmaz*)

Az eddigiektől eltérően a megoldást részfeladatokra osztjuk, érzékeltetve ezzel a hasonló feladatoknál elvégzendő lépéseket.

A Halmaz osztály megtervezése, figyelembe véve a megfogalmazott elvárásokat

Célunk a továbbiakban, hogy a bevezető részben körvonalazott *Halmaz* típusunkhoz olyan operátorfüggvényeket hozzunk létre, amelyek megvalósítják a különféle halmazalgebrai és relációs műveleteket, például a *Pascal* programozási nyelvben megszokottakhoz hasonlóan[1]. Tűzzük ki célul a következő műveletek megvalósítását!

Jel	Művelet	Eredmény
+	Halmazok uniója (egyesítése) `Halmaz a, b; ... a+b;`	Az egyesített halmaz. A közös elemek csak egyszer fordulnak elő az eredményhalmazban!

+	Halmaz és egy egész szám uniója.
	`Halmaz a; int i;`
	`a+i;` illetve `i+a;`
		A halmazba bevesszük a megadott egész számot, és az így kapott halmazt szolgáltatja eredményül. (Kétféle változat az operandusok sorrendjétől függően)

*	Halmazok metszete (közös része) `Halmaz a, b; ... a*b;`	A közös rész, tehát azon elemek halmaza, amelyek mindkét operandusban megtalálhatók.

=	Értékadás.
	`Halmaz a, b; int i;`
	`a=b;` illetve `a=i;`
		Balérték halmaz felveszi a jobbérték halmaz vagy egy egész kifejezéssel meghatározott egyelemű halmaz értékét.

<=	Részhalmaz-vizsgálat.
	`int i; Halmaz a, b;`
	`i = (a <= b);`
		Igaz értéket ad, ha *a* részhalmaza *b*-nek.

>=	Részhalmaz-vizsgálat.
	`int i; Halmaz a, b;`
	`i = (b >= a);`
		Igaz értéket ad, ha *a* részhalmaza *b*-nek.

<	Eleme reláció vizsgálata (Pascal **In** művelet).
	`int i,j; Halmaz a;`
	`j = i < a;`
		Igaz értéket ad, ha *i* eleme *a*-nak.

<<	Adatfolyamba való kiíratás.
	`Halmaz a; ostream os;`
	`os << a << "valami";`
		Visszaadja az adatfolyamot, hogy láncolható legyen a kiíratás.

---

[1] Az egész számok halmazok tárolására választott megoldásunknál van optimálisabb is, például a szabványos Pascal implementációkban szokásos módszer. A halmaztípus jelen megvalósításával az a célunk, hogy könnyen érthető példával szemléltessük a C++ nyelv operátorfüggvényeinek témakörét.

Célszerű, ha a megvalósítás során egymásra támaszkodnak az egyes operátorfüggvények. Például, az összeadás műveletét először kiterjesztjük a *Halmaz+Halmaz* esetre. Ezt követően a *Halmaz*+**int** jellegű „összeadást” oly módon kezeljük, hogy a kérdéses operátorfüggvény az összeadás jobb oldalán álló egyetlen egész számot egyelemű egész tömbbe írja. Ezzel a tömbbel aktiválva a *Halmaz::Halmaz()* konstruktort, létrehozunk egy átmeneti halmazváltozót, amelyet azután hozzáadunk a bal oldali halmazhoz, a már korábban megírt *Halmaz+Halmaz* összeadást megvalósító operátorfüggvény segítségével. Ugyanígy járunk el az **int**+*Halmaz* típusú összeadásnál, de most a bal oldali operandus kerül egy átmeneti, *Halmaz* típusú munkaváltozóba. Így tehát a *Halmaz*::**operator**+*()* függvénynek több változata lesz, és közülük mindig a paraméter-szignatúra alapján választja ki a fordító azt, amelyikkel az adott összeadás megvalósítható.

Ha számba vesszük a fentiekben felsorolt műveleteket, a következőképpen célszerű deklarálni a *Halmaz* típust:

```
// HALMAZ.H
#include <iostream> // az ostream miatt kell
using namespace std;

class Halmaz
{
  private:
   int  *elemek;
   int  darab;
   friend Halmaz operator+(const Halmaz &a, const Halmaz &b);
   friend Halmaz operator+(const Halmaz &a, const int e);
   friend Halmaz operator+(const int e, const Halmaz &a);
   friend Halmaz operator*(const Halmaz &a, const Halmaz &b);
   friend bool   operator<(const int e, const Halmaz &a);
   friend bool   operator<=(const Halmaz &a, const Halmaz &b);
   friend bool   operator>=(const Halmaz &a, const Halmaz &b);
   friend ostream& operator<<(ostream &os, Halmaz &s);
  public:
   Halmaz(int n=0, int *e=NULL);
   ~Halmaz() { if (darab) delete [] elemek; }

   int GetDarab() { return darab; }
   Halmaz& operator=(const Halmaz &s);
   Halmaz& operator=(const int i);
   Halmaz& operator+=(const Halmaz &s);
   bool    operator==(const Halmaz &s);
   bool    operator!=(const Halmaz &s);
};
```

Amint arra utaltunk, az értékadás műveletét csak tagfüggvénnyel valósíthatjuk meg. Továbbá minden olyan esetben, amikor egy operátorban alkalmazott *Halmaz* típusú balérték kifejezés értéke a műveletvégzés folytán biztosan nem változik meg (== != összehasonlítások), illetve amikor kifejezetten a balérték megváltozása a cél (+=), tagfüggvényként valósítjuk meg az operátorfüggvényt. Ha egy kétoperandusú művelet egyik operandusa nem *Halmaz* típusú, vagy a bal oldali operandust nem a **this** mutatón keresztül szeretnénk elérni (hogy értéke ne változzon meg), az operátorfüggvényt a *Halmaz* osztályon kívüli **friend** függvényként állítjuk elő (+ * < <= >= <<).

Annak érdekében, hogy bizonyos halmazműveleteket egyszerűbben lehessen megvalósítani, az eddigi osztálydefiníciót célszerű egy **private** hozzáférésű tagfüggvénnyel bővíteni, amely minden esetben, amikor egy halmaz elemeiben változás áll be, a halmaz elemeit nagyság szerint (újra)rendezi. Ezt a rendezési műveletet az *cstdlib* fejlécállományban deklarált ***qsort()*** szabványos könyvtári függvény felhasználásával egyszerűen megvalósíthatjuk. Legyen a rendező tagfüggvény neve *Halmazrendezo()*! Az elmondottak alapján a osztálydefiníció a következő sorokkal bővül:

```
#include <cstdlib> // a qsort() miatt kell
class Halmaz
{ . . .
  private:
   void Halmazrendezo();
};
```

A halmaz elmeit rendező tagfüggvény és az összehasonlítást végző statikus (modulszintű) függvény az alábbiakban látható:

```
static int cmp(const void *a, const void *b)
// paraméterezés ahogy qsort() igényli
{
  return *(int*)a - *(int*)b;
}

void Halmaz::Halmazrendezo()
{
  qsort(elemek, darab, sizeof(int), cmp);
}
```

Ezt a tagfüggvényt a konstruktorban is aktiválni kell:

```
Halmaz::Halmaz(int n, int *e)
{
  if (n) // Ha nem üres halmaz, helyet foglalunk az elemek-nek
  {      // és e[] tömbből n darab számot a halmazba másolunk.
    darab = n;
    elemek = new int[n];
    for (int i = 0 ; i < n; i++)
       elemek[i] = e[i];
    Halmazrendezo();  // a halmazba másolt számokat sorba rendezzük
  }
  else
  { // üres halmazt hozunk létre
    darab = 0;
    elemek = NULL;
  }
}
```

A kétféle értékadó művelet operátorfüggvénye

Az operátorfüggvények megvalósításának áttekintését kezdjük a kétféle értékadó művelettel! Ugyan már mutattunk példát az értékadásra, de akkor nem adtunk magyarázatot az első **if** utasítással végzett vizsgálatra: ha a bal oldali halmaz nem üres, akkor a jobbérték halmaz bemásolása előtt fel kell szabadítanunk a korábbi elemek által lefoglalt memóriát. További kiegészítésként a művelet befejezésekor elvégezzük az elemek sorba rendezését:

```
Halmaz& Halmaz::operator=(const Halmaz &s)
{
  if (darab) delete [] elemek;  // a korábbi elemek[] eldobása

  darab = s.darab;              // számosság másolása
  if (!s.darab)
    elemek = NULL;              // az új halmaz üres halmaz
  else                          // az új halmaz nem üres, így
  {                             // elemek[]-nek helyet kell
    elemek = new int[darab];    // foglalni, utána másolunk
    for (int i = 0; i < darab; i++)
       elemek[i] = s.elemek[i];
  }
  Halmazrendezo();              // a biztonság kedvéért rendezünk
  return *this;                 // *this a visszatérési érték
}
```

Az értékadás kötelezően tagfüggvény. A **this** implicit mutató azt a balérték halmazt címzi meg, amelynek értéket szeretnénk adni, míg a fenti operátorfüggvény *s* paramétere az értékadás jobb oldalán álló halmazkifejezés. Semmi más dolgunk nincs, mint – a bal oldali halmaz elemei által korábban foglalt memória felszabadítása után – az *s.elemek[ ]* tömböt a *this->elemek[ ]* tömbbe másolni. Ez utóbbit a **new** operátorral hozzuk létre, az *s.elemek[ ]* tömbbel egyező mérettel. Végül a másolás után az elemeket rendezzük a *Halmazrendezo()* tagfüggvény hívásával, hogy mindig, amikor értékadás történik, a rendezettség biztosított legyen, ugyanis erre a későbbiekben számítani fogunk. Amennyiben igazodni szeretnénk a C++ nyelv beépített operátorainál megszokottakhoz, az értékadás művelete visszatérési értékül a balérték kifejezést kell adja – így operátorfüggvényünket a **return *this;** utasítással zárjuk. (Emlékeztetőül: A C++ nyelvben az értékadás műveletének fő hatása az, hogy az értékadás kifejezés értéke megegyezik az értékadás jobbérték kifejezésének értékével, mellékhatása pedig az, hogy a balérték kifejezésként adott változóba bemásolódik a jobbérték kifejezés tartalma. Fenti operátorfüggvényünk is ennek megfelelően működik.)

Az értékadás másik formája az, amikor egyetlen egy egész számból álló, egyelemű halmazt hozunk létre:

```
Halmaz& Halmaz::operator=(const int i)
{
  if (darab) delete [] elemek;
  darab = 1;
  elemek = new int[darab];
  elemek[0] = i;
  return *this;
}
```

A megoldás emlékeztet a konstruktorra, de itt explicit módon megadjuk a számosságot (*darab = 1*) és az egyetlen halmazelemet. Mivel csak egyelemű halmaz jön létre, rendezni nem kell.

Az "eleme" művelet megvalósítása (<)

A következő operátorunk az "eleme-e" reláció vizsgálata, melynek működése a Pascal nyelv **In** operátorának felel meg. A művelet jobb oldali operandusa egy halmaz, bal oldali operandusa pedig egy egész szám. A reláció igaz, ha a kérdéses egész szám eleme az adott halmaznak, egyébként hamis. Ezt a műveletet a < jelhez rendeljük, amely emlékeztet a matematikában használatos ∈ jelre ("eleme" reláció jele).

```
bool operator<(const int e, const Halmaz &a) // eleme-e? vizsgálat
{
  for (int i = 0; i < a.darab; i++)
     if (e == a.elemek[i]) return true;
  return false;
}
```

A megvalósítás igen egyszerű: amint az *e* egész számot megtaláljuk az *a* halmazban, **true** értékkel visszatérhetünk. Ha a kereső ciklusban nem találjuk meg az egész számot a halmazban, akkor a művelet eredménye **hamis**.

Halmazok egyesítése, uniója (+)

Az egyik alapvető halmazalgebrai művelet a halmazok egyesítése, az unióképzés, melyet a + operátorhoz rendelünk. A feladat igen egyszerű: a két összeadandó halmaz elemeit egy közös tömbbe kell másolnunk, figyelembe véve a halmazalgebra szabályait. Az uniót képző mindkét halmazban előforduló azonos elemek az egyesített halmazban csak egyszer szerepelhetnek, így a közös halmazba való másoláskor erről nem szabad megfeledkeznünk. Az egyesített halmaz számossága sem egyszerűen csak a két halmaz számosságának az összege, hiszen a két halmazban szereplő azonos elemeket csak egyszer kell számbavenni. A megvalósítás során már támaszkodunk az imént definiált "eleme-e" relációs operátorra és a halmazértékadás műveletre.

```
Halmaz operator+(const Halmaz &a, const Halmaz &b) // unióképzés
{
  int db;
  int n = a.darab + b.darab;    // az unióhalmaz maximális számossága
  Halmaz eh;                    // az eredményhalmaz, egyelőre üres
  if (n)  // ha nem két üres halmazt adunk össze:
  {
    int *celemek=new int[n];    // az unióhalmaz elemeit rakjuk ide

    // az első halmaz elemeit mind bemásoljuk
    for (db = 0; db < a.darab; db++)
      celemek[db] = a.elemek[db];

    // a másik halmazból csak azokat az elemeket másoljuk be,
    // amelyek az első halmazban nem szerepeltek
    for (int i = 0; i < b.darab; i++)
      if (!(b.elemek[i] < a))  //  az eleme reláció alkalmazása
        celemek[db++] = b.elemek[i];

    // db tartalmazza a végső számosságot
    // db és celemek[] segítségével létrehozzuk az eredményhalmazt
    eh=Halmaz(db, celemek);
    delete [] celemek;          // celemek[] tömböt már eldobhatjuk
  }
  return eh;
}
```

Az unióképzés eredményeként született elemek a *celemek[]* egész tömbbe kerülnek, az eredményhalmaz számosságát pedig az *db* változó tárolja. Ezekből a *Halmaz* típus konstruktorával képezzük az eredményhalmazt, a *Halmaz c(db,celemek)* utasítással. Az így létrejövő automatikus (lokális) *eh* halmazváltozó értékét adjuk vissza függvényértékként a **return** utasításban.

A + operátorhoz még egy további értelmezést rendelhetünk: egy halmaz elemkészletének bővítését egyetlen egy egész számmal. Célszerű, ha ez a fajta „összeadás" nem függ az operandusok sorrendjétől, azaz létezik *Halmaz*+**int** és **int**+*Halmaz* változata is. Ez azt jelenti, hogy az **operator+**() függvénynek a meglévőn kívül még két további, különböző paraméter-szignatúrájú változatát kell elkészítenünk. A megvalósítás során a már meglévő operátorfüggvényekre támaszkodunk. Az összeadás *Halmaz*+**int** szignatúrájú változatát az unióképzésre vezetjük vissza úgy, hogy az egész számból egyelemű halmazt készítünk, és alkalmazzuk az unió műveletet:

```
Halmaz operator+(const Halmaz &a, const int e)
{
  Halmaz b;
  b=e;              // egyelemű halmaz létrehozása
  return a+b;       // visszavezetjük halmazösszeadásra
}
```

Az **int**+*Halmaz* szignatúrájú változatot pedig az imént elkészítettre vezetjük vissza, az operandusok sorrendjének felcserélésével:

```
Halmaz operator+(const int e, const Halmaz &a)
{
   return a+e;     // int+Halmaz-t visszavezetjük Halmaz+int-re
}
```

A += művelet értelmezésének kiterjesztése

Tekintve, hogy a += operátor esetében a bal oldali operandust (ami balérték kifejezés, azaz változó kell legyen) módosítani szeretnénk, illetve valójában értékadási műveletet kell végeznünk, az operátorfüggvényt tagfüggvényként valósítjuk meg. Tehát egy *a+=s* típusú halmazművelet végrehajtása egyenlő a ***this=*this**+*s* végrehajtásával, ahol **this** az *a* halmazra mutat. Ezzel meg is adtuk a megoldás alapgondolatát.

```
Halmaz& Halmaz::operator+=(const Halmaz &s)
{
  Halmaz a(darab,elemek); // a tartalma azonos lesz *this-szel
  a = a + s;              // összeadás a+=s értelemben
  // eldobjuk *this régi tartalmát
  *this=a;
  // majd az új *this halmazzal térünk vissza
  return *this;
}
```

Halmazok közös része, metszete (*)

Két halmaz közös részének képzéséhez hasonló lépések szükségesek, mint amiket az unió művelet kapcsán megtettünk. Éppen ezért az unióhoz hasonlóan **friend**-ként célszerű megvalósítanunk ezt az operátorfüggvényt. További megfontolásaink már csak a halmazalgebrával kapcsolatosak. A metszethalmaz a két kiinduló halmazból csak azokat az elemeket tartalmazza, amelyek mindkettőben megtalálhatók. Célszerű tehát úgy előállítani két halmaz metszetét, hogy kezdetben üres halmaznak tekintjük a közös részt, majd például sorra vesszük a kisebbik számosságú halmaz elemeit és megvizsgáljuk, hogy megtalálhatóak-e a másik halmazban. (Erre felhasználhatjuk az ∈ reláció vizsgálatára korábban megírt **operator**< *()* operátorfüggvényünket.) Ha egy elem a másik halmazban is megtalálható, felvesszük az eredményhalmazba.

```
Halmaz operator*(const Halmaz &a, const Halmaz &b) // metszetképzés
{
  int k,              // a kisebb számosságú halmaz elemszáma
      n;              // a nagyobb számosságú halmaz elemszáma
  int *celemek;       // a metszethalmaz elemeihez
  Halmaz nagyobb, kisebb;

  // melyik a nagyobb számosságú a két operandus közül
  if (a.darab > b.darab) {
    n = a.darab;
    k = b.darab;
    nagyobb = a;
    kisebb = b;
  }
  else {
    n = b.darab;
    k = a.darab;
    nagyobb = b;
    kisebb = a;
  }
  // a közös részhalmaz nem lehet nagyobb, mint a kisebbik halmaz,
  // így annak elemszámával foglalunk helyet a celemek[] tömb számára
  celemek = new int[k];

  // a kisebbik halmaz minden elemére megvizsgáljuk, hogy az eleme-e a
  // nagyobbik halmaznak, ha igen, a közös rész halmazba kerül
  n = 0; // n-ben számláljuk, hogy hány közös elemet találunk
  for (int i = 0; i < k; i++)
  {
    if (kisebb.elemek[i] < nagyobb)
       celemek[n++] = kisebb.elemek[i];
  }
  Halmaz eh(n,celemek);       // eh-ban a közös rész
  delete [] celemek;
  return eh;
}
```

Részhalmaz-vizsgálat (<=, >=)

Logikai (**bool**) típusú lesz a visszatérési érték, ezért **friend** függvényt írunk. A működés könnyen átgondolható:

1. Az üres halmaz minden halmaznak részhalmaza, ezért ilyen esetben a visszatérési érték **true** kell legyen.
2. Nagyobb számosságú halmaz nem lehet kisebb számosságú halmaz részhalmaza, ezért ilyen esetben a visszatérési érték **false** lesz.
3. Az $A \subseteq B$ reláció (A részhalmaza B-nek) csak akkor igaz, ha A halmaz minden eleme, eleme B halmaznak, azaz minden $a \in A$-ra igaz, hogy $a \in B$.

Komolyabb programozási munkát csak a 3. pont kidolgozása igényel, azonban egy egyszerű **for** ciklussal és az "eleme-e" relációs operátor felhasználásával ez is könnyen elvégezhető:

```
bool operator<=(const Halmaz &a, const Halmaz &b)
{
  // az üres halmaz minden halmaznak részhalmaza
  if (!a.darab) return true;

  // a nagyobb számosságú halmaz nem lehet kisebb számosságú
  // halmaz részhalmaza
  if (a.darab > b.darab) return false;

  bool ret=true;
  // a ret csak akkor lesz true, ha a valódi részhalmaza b-nek
  for (int i = 0; i < a.darab; i++)
     ret = ret && (a.elemek[i] < b);
  return ret;
}
```

A fordított reláció vizsgálatát egy egyszerű operanduscserével a fenti függvényre vezetjük vissza:

```
bool operator>=(const Halmaz &a, const Halmaz &b)
{
  return b <= a;
}
```

Halmazok azonosságának és különbözőségének vizsgálata (==, !=)

Az alábbi megoldásokhoz csak annyit érdemes hozzáfűzni, hogy az "egyenlő-e" reláció vizsgálatakor arra támaszkodunk, hogy

- a halmazaink rendezettek, és
- nem tartalmaznak ismétlődő elemeket.

```
bool Halmaz::operator==(const Halmaz &s)
{
  // különböző számosságú halmazok
  if (darab != s.darab) return false;

  // két üres halmaz
  if (!darab) return true;

  // két valódi, azonos számosságú halmaz
  for (int i = 0; i < darab; i++)
      if (elemek[i] != s.elemek[i])
         return false;
  return true;
}

bool Halmaz::operator!=(const Halmaz &s)
{
  return !(*this==s);
}
```

A Halmaz típus működését bemutató főfüggvény

```
// MAIN.CPP
#include <iostream>
using namespace std;
#include "Halmaz.h"

int aelemek[] = {5,4,3};
int main(void)
{
  int i;
  Halmaz a(sizeof(aelemek)/sizeof(int), aelemek);
  Halmaz b,c,d;

  cout << "A Halmaz tipus es operatorainak bemutatasa.\n\n";
  cout << "A definialt halmazok:"<<endl;
  cout << "A halmaz :  " << a <<endl;
  cout << "B halmaz :  " << b <<'\t';
  cout << "C halmaz :  " << c <<'\t';
  cout << "D halmaz :  " << d <<endl;

  cout << "\nUniokepzes es ertekadas: A = A + 12\t\t";
  a = a + 12;
  cout << "A halmaz :  " << a <<endl;
  cout << "\nOsszetett uniokifejezes: B=((B+1)+2)+3\t\t";
  b = ((b + 1) + 2) + 3;
  cout << "B halmaz :  " << b <<endl;

  cout << "\nA szammal vegzett ertekadas: C = 23\t\t";
  c = 23;
  cout << "C halmaz :  " << c <<endl;

  cout << "\nUniomuvelet ertekadassal: B += A \t\t";
  b += a;
  cout << "B halmaz :  " << b <<endl;

  cout << "\nTobbszoros ertekadas: D = C = C + 12\t\t";
  d = c = c + 12;
  cout << "C halmaz :  " << c <<'\t';
  cout << "\t\t\t\t\t\t\tD halmaz :  " << d <<endl;

  if (d == c)
      cout << "D es C halmaz megegyezik (D == C)!\n";
  else
      cout << "D es C halmaz kulonbozik!\n";

  cout << "\nKozosreszkepzes: D = C * A\t\t\t";
  d = c * a;
  cout << "D halmaz :  " << d <<endl;

  if (d <= a)
      cout << "A D halmaz reszhalmaza az A halmaznak (D <= A).\n";

  cout << "\nEleme-e relacio: (12 < D) - 12 eleme D-nek?\t";
  if (12 < d)
     cout << "Igen.\n";
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
A Halmaz tipus es operatorainak bemutatasa.

A definialt halmazok:
A halmaz :  3 4 5
B halmaz :  ures       C halmaz :  ures       D halmaz :  ures

Uniokepzes es ertekadas: A = A + 12           A halmaz :  3 4 5 12

Osszetett uniokifejezes: B=((B+1)+2)+3       B halmaz :  1 2 3

A szammal vegzett ertekadas: C = 23           C halmaz :  23

Uniomuvelet ertekadassal: B += A              B halmaz :  1 2 3 4 5 12

Tobbszoros ertekadas: D = C = C + 12          C halmaz :  12 23
                                              D halmaz :  12 23
```

```
D es C halmaz megegyezik (D==C)!

Kozosreszkepzes: D = C * A                  D halmaz :   12
A D halmaz reszhalmaza az A halmaznak (D<=A).

Eleme-e relacio: (12<D) - 12 eleme D-nek?     Igen.
```

19.2 Saját sztring típus és a hozzá tartozó műveletek definiálása

Tervezzünk és definiáljunk egy Sztring osztályt, amely lehetővé teszi a karaktersorozatok sztringszerű kezelését! (*Sztring*)

Mielőtt hozzálátnánk a feladat megoldásához, értelmezzük azt! A „sztringszerű kezelés" alatt azt értjük, hogy az osztály támogatja tetszőleges hosszúságú karaktersorozatok használatát, biztosítja az indexelést, a sztringek összefűzését, valamint a sztring-összehasonlítást. Bár ezek a műveletek az ANSI C szabványos könyvtári függvényei révén rendelkezésünkre állnak (*cstring*), mégis érdemes a C++ nyelv lehetőségeinek felhasználásával is megvalósítanunk őket, hiszen a sztringműveleteket végző programok gyors fejlesztéséhez a *BASIC* vagy *Turbo Pascal* programozási nyelvekben megszokott műveletek nagy segítséget nyújthatnak[2].

Alapvetően most is az operátor-túlterhelés (*operator overloading*) eszközeit fogjuk használni. A korábbi halmazpéldánk kapcsán bemutatott lehetőségekhez képest újdonságként hat az indexelő, a függvényaktivizáló valamint a típusmódosító operátor értelmezésének kiterjesztése az új *Sztring* típusunkra. Az alábbi táblázatban összefoglaltuk a *Sztring* típussal újraértelmezett műveleti jeleket.

Jel	Művelet	Eredmény
+	Sztringek összefűzése (konkatenálása) `Sztring a, b; ... a+b;`	Az egyesített sztring, amelynek első részét az *a*, második részét a *b* sztring alkotja.
+	Sztring és c-karaktersorozat összefűzése (konkatenálása) `Sztring a; ... a+"valami"`	Az egyesített karaktersorozat, melynek első részét az *a Sztring*, második részét pedig a c-karaktersorozat alkotja.

=	Értékadás sztringnek.
	`Sztring a, b; char c[]="xx";`
	`a=b;` illetve `a=c;`
		Balérték sztring felveszi a jobbérték sztring (kifejezés) vagy a **char**[] karaktersorozat értékét.

[]	Sztring indexelése.
	`Sztring a("alma"); int i=1;`
	`char ch = a[i];`
		A sztring *i*-edik karakterét adja vissza.

---

[2] Megjegyzendő, hogy az ANSI C++ nyelvi implementáció a rendelkezésünkre bocsát olyan osztálykönyvtárat, amelyben a sztringek valamilyen megvalósítása már létezik. C++ programjaink tervezése során tehát célszerű ezzel a lehetőséggel is számolni. Jelen példával is, hasonlóan a korábbi halmaztípushoz, a C++ operátorfüggvények témakörét szeretnénk jól érthetően illusztrálni.

`()`	Aktivizáló operátor.
	`Sztring a("alma");`
	`char ch = a();`
		Minden aktivizáláskor a sztring soronkövetkező elemét adja vissza, ha a sztring végére ért, kezdi elölről (iterátor).

`(char*)`	Típuskonverziós operátor
	`Sztring a("alma"); char *p;`
	`p = (char*)(a);`
		A sztringet **char*** típussá (karaktertömbre mutató pointerré) alakítja.

`==`	Egyenlőség vizsgálata.
	`bool r; sztring a, b;`
	`r = (a == b);`
		Igaz értéket ad, ha az *a* sztring azonos a *b*-vel.

`!=`	Egyenlőtlenség vizsgálata.
	`bool r; sztring a, b;`
	`i = (b != a);`
		Igaz értéket ad, ha az *a* sztring különbözik a *b*-től.

`<`	Kisebb reláció vizsgálata.
	`int i; Sztring a,b;`
	`j = (a < b);`
		Igaz értéket ad, ha *a* kisebb, mint *b*, azaz az angol ABC-ben előbb szerepel.

`>`	Nagyobb reláció vizsgálata.
	`int i; sztring a,b;`
	`j = (a > b);`
		Igaz értéket ad, ha *a* nagyobb, mint *b*, azaz az ABC-ben később szerepel.

`<<`	Output adatfolyamba való kiíratás.
	`Sztring a; ostream os;`
	`os << a << "valami";`
		Kiírja a sztringet, és visszaadja az adatfolyamot, hogy láncolható legyen a kiíratás.

`>>`	Olvasás input adatfolyamból.
	`Sztring a; istream is;`
	`is >> a;`
		Tetszőleges szöveget olvas be a bemenő steramből. Visszaadja az adatfolyamot, hogy láncolható legyen a beolvasás.

Ezek után nézzük a vonatkozó osztálydefiníciót!

```
//SZTRING.H
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cctype>
#include <iostream>

class Sztring {
  private:
   char *szoveg;                          // Sztring értéke
   int  hossz;                            // Sztring hossza
   int  index;                            // aktuális elem
  public:

   // Konstruktorok
   Sztring(char *s = NULL);               // tömbbel hozza létre
   Sztring(Sztring &s);                   // másoló konstruktor
   // Destruktor (inline):
   ~Sztring() {
      if (szoveg != NULL) delete [] szoveg;
      szoveg = NULL; hossz = 0;
   }
```

```
    // Tagfüggvények:
    int  Hossza(void) { return hossz; }
    void Nagybetusit(void);
    void Kisbetusit(void);

    // Értékadó operátorok:
    Sztring& operator=(Sztring& s);     // Sztring = Sztring értékadás
    Sztring& operator=(char* p);        // Sztring = char[] értékadás

    // Sztringösszefűző operátorok:
    Sztring  operator+(Sztring &s);      // Sztring+Sztring összefűzés
    Sztring  operator+(char* p);         // Sztring+char[] összefűzés

    // Relációs operátorok (inline)
    bool operator==(Sztring &s) { return !strcmp(szoveg,s.szoveg); }
    bool operator!=(Sztring &s) { return strcmp(szoveg,s.szoveg)!= 0; }
    bool operator<(Sztring &s)  { return 0 > strcmp(szoveg,s.szoveg); }
    bool operator>(Sztring &s)  { return 0 < strcmp(szoveg,s.szoveg); }

    // Indexelő operátor
    char& operator[](const int ind);  // kötelezően tagfüggvény

    // "Következő eleme" (iterátor) operátor
    char operator()(); // kötelezően paraméter nélküli

    // Típuskonverziós operátor
    operator char*(); // kötelezően paraméter és típus nélküli

  private:
    // Indexhatár-ellenőrzés privát függvénnyel
    bool HatarOk(int ind);    // igaz, ha ind index lehet

    // String kiiratása, beolvasása
    // kiirás streambe
    friend ostream& operator<<(ostream &os, Sztring &s);
    // beolvasás streamből
    friend istream& operator>>(istream &is, Sztring &s);
};
```

A *Sztring* típus három privát hozzáférésű adataggal rendelkezik: a tárolt szöveg hossza (egész szám), a tárolt szöveg (karakterpointer) és egy harmadik egész szám, amely az iterátorfüggvény által használt index.

A *Sztring* osztály számos tagfüggvényt tartalmaz, ezek többsége a fenti táblázatnak megfelelő operátorfüggvény. Ezek, akárcsak a konstruktorok, nyilvános elérésű függvények. Az adatfolyamokkal kapcsolatos operátorfüggvények, akárcsak a korábbi *Halmaz* osztály esetében, itt is **friend** függvények.

Konstruktorok

Részletes tárgyalásunkat kezdjük a konstruktorokkal! Amint az osztály deklarációjából kiderül, kétféle konstruktort készítettünk. Az egyik **char**[] karaktertömbből (hagyományos C-karaktersorozatból) hoz létre *Sztring* típusú objektumot. Az osztálydeklarációban a konstruktor karaktertömbre mutató pointer paraméteréhez a *NULL* értéket rendeltük alapértékként. Ennek következtében, ha argumentum nélkül definiálunk egy *Sztring* objektumot, az objektumpéldányában a *szoveg* adattag *NULL* értéket kap, így a sztring karakterei számára nem foglalunk memóriát. Ez azt jelenti, hogy az argumentum nélkül definiált *Sztring* objektumok automatikusan üres sztringek lesznek. Ha egy karaktertömbbel aktivizáljuk ezt a konstruktort, akkor először az ***strlen()*** könyvtári függvény segítségével maghatározzuk a karaktersorozat hosszát, majd a karaktereknek és a karakterfüzér végét jelző '\0' bájtnak helyet foglalunk, végül pedig feltöltjük a sztringobjektumot az inicializáláshoz használt karaktertömb elemeivel.

```
Sztring::Sztring(char *s) // tömbből sztringet készítő konstruktor
{
   if (s != NULL)
   { // ha van karaktertömb-paraméter
      if (!(hossz = strlen(s)))
      { // ha üres sztringgel inicializálunk ""
        hossz = index = 0, szoveg = NULL;
        return;
      }
      szoveg = new char[hossz+1]; // +1 hely EOS-nek
      index = 0;
      strcpy(szoveg, s);
   }
   else
        hossz = index = 0, szoveg = NULL;
}
```

Hasznos lehetőség, ha egy új objektumpéldányt egy már létező objektumpéldány másolataként hozhatunk létre. Ilyen másoló konstruktor a *Sztring* osztály másik konstruktora:

```
// Sztringből Sztringet készítő (copy) konstruktor
Sztring::Sztring(Sztring &s)
{
  if (s.hossz)
  {
     hossz = s.hossz;
     szoveg = new char[hossz+1]; // +1 hely EOS-nek
     index = 0;
     strcpy(szoveg, s.szoveg);
  }
   else
        hossz = index = 0, szoveg = NULL;
}
```

A destruktor dolga pusztán a sztring karakterei által foglalt memória-terület felszabadítása, éppen ezért ezt **inline** formában valósítottuk meg (lásd az osztálydeklarációban).

Értékadás objektummal és karaktersorozattal

Az értékadó operátorok megvalósítása sokban emlékeztet a konstruktorokéhoz. Először tekintsük a *Sztring-Sztring* értékadást!

```
// Sztring = Sztring értékadás
Sztring& Sztring::operator=(Sztring& s)
{
  if (hossz) delete [] szoveg;     // a régi tartalom törlése
  if (s.hossz)
  {
     hossz = s.hossz;
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, s.szoveg);
  }
   else
        hossz = index = 0, szoveg = NULL;
  return *this;
}
```

A másoló konstruktorhoz képest itt annyi a különbség, hogy a **this** által mutatott balérték kifejezés korábbi értékét el kell dobni, illetőleg a másolás után a ***this** referenciával térünk vissza, (hogy láncolható legyen a sztringértékadás művelete).

Teljesen analóg módon készült a C-stílusú karaktersorozattal végzett értékadás művelete:

```
Sztring& Sztring::operator=(char *p)  // Sztring = char* értékadás
{
  if (hossz) delete [] szoveg;  // a régi tartalom törlése
  hossz = strlen(p);
  if ((p != NULL) && hossz)
  {
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, p);
  }
  else
     hossz = index = 0, szoveg = NULL;
  return *this;
}
```

Sztringek összefűzése (konkatenálás)

A sztringek összefűzésének műveletét a + operátorhoz rendeljük. A halmaz típus kapcsán bemutatott megoldástól eltérően most tagfüggvényként valósítjuk meg a vonatkozó operátorfüggvényeket. Először tekintsük a *Sztring+Sztring* művelet megvalósítását!

```
Sztring Sztring::operator+(Sztring& s)
{
  char *kozos_szoveg = NULL;
  int kozos_hossz = hossz + s.hossz;
  Sztring ret;

  if (hossz && s.hossz) // ha egyik sztring sem üres
  {
     kozos_szoveg = new char[kozos_hossz + 1];
     strcpy(kozos_szoveg,szoveg);
     strcat(kozos_szoveg,s.szoveg);
  }
  else
     if (!hossz && s.hossz) // csak a bal oldali Sztring 0 hosszú
     {
        kozos_szoveg = new char[s.hossz + 1];
        strcpy(kozos_szoveg,s.szoveg);
     }
     else  // ellenkező esetben
        return *this;

  ret = kozos_szoveg; // Sztring = char* értékadás
  if (kozos_szoveg)
     delete [] kozos_szoveg;
  return ret;
}
```

Az összeadás művelet bal oldali operandusát a tagfüggvényként definiált operátorfüggvény ***this** implicit paramétere szolgáltatja, a jobb oldali operandus pedig a függvény *s* paraméterben jelenik meg.

A két sztring együttes hosszának megfelelő memória-terület lefoglalása után a tényleges összefűzést az ***strcat()*** könyvtári függvény segítségével végezzük. A művelet eredményét – a *Halmaz* típus kapcsán leírt meggondolások alapján – a másoló konstruktor által létrehozott *Sztring* értékként szolgáltatjuk.

A fenti összefűzés megvalósítása után egyszerűvé vált a *Sztring*+**char***[]* művelet operátorfüggvényének elkészítése, hiszen a konstruálást követően két *Sztring* típusú karaktersorozatot fűzünk össze:

```
Sztring Sztring::operator+(char* p)
{
  Sztring ret(p);
  return *this+ret;
}
```

Összehasonlító (relációs) műveletek

A relációs műveletek operátorfüggvényeit az ***strcmp()*** szabványos könyvtári függvény „megfejelésével” készítettük el. Mivel ezek az operátorfüggvények igen egyszerűek, **inline** tagfüggvényként (azaz bináris makróként) definiáltuk őket az osztálydefiníciós blokkban. Ezen operátorfüggvények dolga mindössze annyi, hogy a könyvtári karaktersorozat-összehasonlító függvény visszatérési értékét a relációs műveleteknél szokásos logikai értékké alakítsák.

```
   bool operator==(Sztring &s) { return !strcmp(szoveg,s.szoveg);     }
   bool operator!=(Sztring &s) { return strcmp(szoveg,s.szoveg)!= 0; }
   bool operator<(Sztring &s)  { return 0 > strcmp(szoveg,s.szoveg); }
   bool operator>(Sztring &s)  { return 0 < strcmp(szoveg,s.szoveg); }
```

A Sztring osztály be- és kiviteli műveletei

A kiíró művelet megvalósítása során induljunk ki a *Halmaz* típusnál alkalmazott operátorfüggvényből! Jelen esetben is „barátként" definiáljuk az operátorfüggvényt, amely első argumentumként egy *ostream* típusú adatot vár (ebbe a *stream*be írunk), és visszatérési értékként is ezt az adatfolyamot adja (miután beleírtunk), hogy láncolható legyen a kiíratás. A korábbi megoldáshoz képest annyi a módosítás, hogy a *Sztring* osztály *szoveg* tagját közvetlenül is kiírathatjuk az adatfolyamba (a << operátor **char*** típusra értelmezett változatával):

```
ostream& operator<<(ostream &os, Sztring &s) // kiírás streambe
{
   if (s.hossz) os << s.szoveg;
   return os;
}
```

A beolvasás >> műveletéhez tartozó tagfüggvény paramétereit és visszatérési értékét a szokásos módon definiáltuk. Az operátorfüggvényben először egy karaktertömbbe (pufferbe) olvassuk a szöveget, majd a *Sztring*=**char*** értékadással a *Sztring* típusú *s* referencia-paraméternek adjuk értékül.

```
istream& operator>>(istream &is, Sztring &s) // beolvasás streamből
{
  char buffer[256];
  is.getline(buffer,256);
  s = buffer;
  return is;
}
```

Indexelő operátor

Miután megvalósítottuk az alapvető sztringműveletekhez (értékadás, összefűzés, kiíratás, beolvasás, relációs műveletek) tartozó operátorfüggvényeket, készítsük el a karaktertömbökkel való kompatibilitást biztosító *indexelő* műveletet!

Az indexelő operátort (*[]*) tagfüggvényként kell megadnunk, mely **char&**, típusú visszatérési értékkel rendelkezik, így egy indexelt sztring tetszőleges karakterét olvashatjuk vagy írhatjuk a segítségével. Annak kivédésére, hogy hibás indexelés ne okozzon súlyos futási hibát, készítettünk egy privát hozzáférésű tagfüggvényt (*HatarOk()*), amellyel az indexelt sztring aktuális indexhatárait ellenőrizhetjük.

```
bool Sztring::HatarOk(int ind)
{
   return 0 <= ind && ind < hossz;
}
```

Indexhatár-túllépés esetén (az egyszerűsége kedvéért) a szabványos hibakimenetre egy üzenetet írunk, és visszatérési értékként a statikus élettartamú *semmi* karakter referenciáját szolgáltatjuk. Helyes indexelés esetén a sztring értékét tároló karaktertömb megfelelő elemét adja vissza a függvény.

```
char& Sztring::operator[](const int ind)
{
  static char semmi='?'; // a referncia-függvényérték hibás esetekre

  if (HatarOk(ind)) return szoveg[ind];
  else
  {
    cerr << "\nHibas indexeles: az " << ind
         << " index kivul esik a hatarokon.\n";
    return semmi;
  }
}
```

Saját típuskonverziós operátor készítése

Az indexelhetőségen túl fontos feladat a karaktertömbökkel való kompatibilitás szempontjából a típuskonverzió. A **char*** ⇒ *Sztring* átalakítás a megfelelő konstruktor, illetve értékadó operátor révén megoldott, de a sztringekre mutató karakterpointerek előállításához saját típuskonverziós operátort kell írnunk.

Az elkészítendő operátorfüggvény szintén kötelezően tagfüggvény kell legyen, és nem rendelkezhet semmilyen paraméterrel.

```
Sztring::operator char*() // típuskonverziós operátor
{
   if (!hossz)
      return NULL;
   else
      return szoveg;
}
```

A típuskonverziós operátorfüggvény azonosítójának meg kell egyeznie azzal a típussal, amibe konvertálni szeretnénk – esetünkben ez a **char***. Az operátorfüggvényünknek semmi más dolga nincs, mint a karaktersorozatot tároló tömb kezdőcímét (*szoveg*) visszaadni. (Egyszerűsége révén ez a függvény akár **inline** függvény is lehetne.)

Az iteráló operátor

Kényelmi szolgáltatásként definiáljunk még egy operátorfüggvényt, amely minden egyes hívás alkalmával az adott sztring következő karakterét adja vissza! Ezt a műveletet rendeljük a függvényaktiváló operátorhoz! Az ilyen operátort iteráló operátornak szokás nevezni, melynek megvalósítása:

```
char Sztring::operator()()  // kötelezően paraméter nélküli
{
  if (!hossz) return 0;     // ha a sztring 0 hosszú
  if (index < hossz)
    return szoveg[index++];
  else
    return szoveg[index = 0];
}
```

A függvényhívás *()* operátorához rendelt operátorfüggvény kötelezően paraméter nélküli. Az *index* adattagban tároljuk az aktuális karakterpozíciót. Ha az indexeléssel a sztring végére érünk, kezdjük elölről. Az iterátorfüggvény használatát az alábbi programpélda szemlélteti:

```
#include <iostream>
using namespace std;
#include "Sztring.h"
enum {PONT = '.', VESSZO = ',', IDEZO = '\"'};
int main(void)
{
  Sztring d(" Szokozzel felulirjuk a ., es \" irasjeleket");
  cout << "Az eredeti sztring: \t" << d << '\n';
  for (int i = 0; i < d.Hossza(); i++)
  {
    char ch = d(); // d sztring következő karaktere
    switch (ch)
    {
       case PONT:  case VESSZO:  case IDEZO:
          d[i] = ' ';
          break;
    }
  }
  cout << "Az eredmenysztring: \t" << d << '\n';
  return 0;
}
```

A Sztring típus további tagfüggvényei

A *Sztring* osztályunk még további három nyilvános hozzáférésű tagfüggvénnyel rendelkezik. Az egyikkel (*Hossza()*) a sztring karaktereinek száma kérdezhető le – ezt használtuk a fenti példaprogramban is. A másik két tagfüggvénnyel a sztring karaktereit kisbetűssé, vagy nagybetűssé alakíthatjuk. Megvalósításukhoz a *cctype* fejlécállományban deklarált ***toupper()*** és ***tolower()*** függvényeket használtuk.

```
void Sztring::Kisbetusit(void)
{
  for (int i = 0; i < hossz; szoveg[i++] = tolower((*this)()));
  index = 0;
}

void Sztring::Nagybetusit(void)
{
  for (int i = 0; i < hossz; szoveg[i++] = toupper((*this)()));
  index = 0;
}
```

Vegyük észre, hogy a *Sztring* típus *Kisbetusit()* és *Nagybetusit()* tagfüggvényei, illetve operátorfüggvényeinek egy része (például a relációs műveletek) pusztán egyszerűsített írásmódot jelentenek a hagyományos könyvtári függvények hívásával szemben. Ezeket azonban megfontoltan kell használni, mert ugyan a C++ forráskódunk sokkal elegánsabb a kiterjesztett jelentésű operátorok alkalmazása révén, de a futtatható programkód hatékonysága rosszabb, mintha magunk C nyelven kódolnánk az egyes műveleteket.

A *Sztring* osztály használatát szemléltető programmodul:

```
//SZTRING.CPP
#include <iostream>
using namespace std;
#include "Sztring.h"

enum {PONT = '.', VESSZO = ',', IDEZO = '\"'};
int main(void)
{
  cout << "\nSajat Sztring tipus lehetosegeinek bemutatasa:\n\n";

  Sztring a("Alfa ");
  Sztring b("Beta ");
  cout << "Sztring(char*) konstruktor hasznalata:\n";
  cout << "\tA = "<< a << "\t\t" << "B = "<< b << endl;

  Sztring c;
  Sztring d;
  cout << "\nSztring default konstruktor alkalmazasa:\n";
  cout << "\tC = "<< c << "\t\t" << "D = "<< d << endl;

  Sztring e(a);
  cout << "\nSztring(Sztring) konstruktor alkalmazasa E(A):\n";
  cout << "\tE = "<< e << endl;

  cout << "\nSztring+Sztring osszefuzes C = A + B :\n";
  c = a + b;
  cout << "\tC = "<< c << endl;

  cout << "\nSztring+char* osszefuzes D = C+ |, \"Gamma\".|:\n";
  d = c + ", \"Gamma\". ";
  cout << "\tD = "<< d << endl;

  cout << "\nD-bol az irasjelek torlese az indexelo es az"
          "iterator muveletekkel:\n";
  for (int i = 0; i < d.Hossza(); i++)
  {
    char ch = d(); // d Sztring következő karaktere
    switch (ch) {
       case PONT: case VESSZO: case IDEZO:
          d[i] = ' ';  break;
    }
  }
  cout << "\tD = "<< d << endl;

  cout << "\nTipuskonverzio bemutatasa strlen(A):\n";
  cout<<"A hossza "<<strlen(a)<<endl;
  if (a < b) cout << "Az " << a << " Sztring kisebb, mint a "
                  << b << " Sztring."<<endl;

  cout << "\nAdjon meg egy szot: ";
  cin >> e;
  e.Kisbetusit();
  cout << "Ez csupa kisbetuvel: " << e << '\n';
  e.Nagybetusit();
  cout << "Ez csupa nagybetuvel: " << e << '\n';

  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Sajat Sztring tipus lehetosegeinek bemutatasa:

Sztring(char*) konstruktor hasznalata:
        A = Alfa                B = Beta

Sztring default konstruktor alkalmazasa:
        C =             D =

Sztring(Sztring) konstruktor alkalmazasa E(A):
        E = Alfa

Sztring+Sztring osszefuzes C = A + B :
        C = Alfa Beta

Sztring+char* osszefuzes D = C+ |, "Gamma".|:
        D = Alfa Beta , "Gamma".

D-bol az irasjelek torlese az indexelo es az iterator muveletekkel:
        D = Alfa Beta    Gamma

Tipuskonverzio bemutatasa strlen(A):
A hossza 5
Az Alfa  Sztring kisebb, mint a Beta  Sztring.

Adjon meg egy szot: Nata
Ez csupa kisbetuvel: nata
Ez csupa nagybetuvel: NATA
```

19.3 A szabványos sablontár (STL) alkalmazása

Ebben a részben néhány korábban megvalósított példaprogram megoldását átfogalmazzuk, a szabványos sablontár osztályainak felhasználásával. (A javasolt elméleti könyv függelékében találunk kiegészítő információkat.)

A 17. fejezet *Tan_02* feladatának megoldásában a **char *** típusú C-sztringek helyett, használjuk a C++ nyelv ***string*** osztályát! (*Tan_02_String*)

A ***string*** osztály alkalmazása esetén a *string* fejlécállományt kell a programunkba építeni. Ez az osztály operátorokat biztosít a leggyakoribb karakterlánc-műveletekhez (értékadás, összefűzés, összehasonlítása, kiírás, bevitel stb.) Ezek felhasználásával a *Tanulo* osztály kezelése sokkal egyszerűbbé válik. A *Tanulo* és a *Lista* osztályok módosított változatai:

```
//OSZTALY.H
#include <string>
using namespace std;

class Tanulo
{
  public:
    string vnev;
    string knev;
    double atlag;
    int osztondij;
    Tanulo *kov;
    Tanulo *elozo;
  public:
    Tanulo(const string & vnev0, const string & knev0,
           double atlag0, int osztondij0);
    ~Tanulo();
};

class Lista
{
  private:
    Tanulo *elso;
    Tanulo *utolso;
    int MaxOsztondij;
    double TanulokAtlaga;
    int Letszam;
  public:
    enum {Elore, Hatra};
    Lista();
    ~Lista();
    void AddTanulo(const string & vnev0, const string & knev0,
                   double atlag0, int osztondij0, int irany);
    void KiirElore();
    void Szamol();
    double GetAtlag(){ return TanulokAtlaga;}
    int GetMaxOsztondij() {return MaxOsztondij;}
    void DelTanulo(const string & vnev0, const string & knev0);
    int GetLetszam(){return Letszam; }
};
```

Az *OSZTALY.CPP* forrásállományból csak a megváltozott *Tanulo::Tanulo()*, *Lista::AddTanulo()* és *Lista::DelTanulo()* tagfüggvényeket nézzük meg közelebbről. Mindhárom tagfüggvényben a neveket tároló adattagok és/vagy paraméterek típusát cseréltük le a ***string*** típusra.

```
Tanulo::Tanulo (const string & vnev0, const string & knev0,
                double atlag0, int osztondij0)
{
  vnev = vnev0; knev = knev0;
  atlag = atlag0; osztondij = osztondij0;
  elozo = kov = NULL;
}

void Lista::AddTanulo (const string & vnev0, const string & knev0,
                       double atlag0, int osztondij0, int irany)
{
  Tanulo *p;
  Letszam++;           //  létszám növelése
  if (elso == NULL) // ez lesz az első
  {
    elso = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    utolso = elso;
  }
  else if (irany == Lista::Elore)  // első elé fűz
    p = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    p->kov = elso;
    elso->elozo = p;
    elso = p;
  }
  else  // az utolsó mögé fűz
  {
    p = new Tanulo(vnev0, knev0,atlag0,osztondij0);
    utolso->kov = p;
    p->elozo = utolso;
    utolso = p;
  }
}

void Lista::DelTanulo(const string & vnev0, const string & knev0)
{
  Tanulo *p, *p1, *p2;
  p = elso;
  while(p != 0)
  {
    if (p->vnev == vnev0 && p->knev == knev0 )
    { Letszam--;
      p1 = p->elozo;
      p2 = p->kov;
```

```
        if (p1 == 0)
        {
          elso = p2;
          if(p2 != 0)
          {
            p2->elozo = 0;
          }else utolso = p2;
        }
        else
        {
          p1->kov = p2;
          if(p2 != 0)
          {
            p2->elozo = p1;

          }
          else
          {
             utolso = p1;
          }
        }
        p->kov = 0; p->elozo = 0;
        delete p;
        break;
      }
      p = p->kov;
    }
}
```

A főmodulban szintén csak néhány típust kellett lecserélnünk, azonban a működés változatlan maradt:

```
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
int main()
{ int n,osztondij0;
  char ch;
  Lista *p;
  string vnev0, knev0;
  double atlag0;
  p = new Lista();
  cout << "Tanulok szama: "; cin >> n;
  if ( n > 0)
  {
   for( int i = 0; i< n ; i++)
   { cout << "Vezeteknev: ";  cin >> vnev0;
     cout << "Keresztnev: ";  cin >> knev0;
     cout << "Atlag      : ";  cin >> atlag0;
     cout << "Osztondij : ";  cin >> osztondij0;
     cout << endl;
     p->AddTanulo(vnev0,knev0,atlag0,osztondij0, Lista::Hatra);
   }
```

```
    p->KiirElore();
    if (p->GetLetszam())
    {
      cout << endl << "Adat torlese (i,I): "; cin >> ch;
      if( ch == 'i' || ch == 'I')
      {
        cout << "Torlendo vezeteknev: "; cin >> vnev0;
        cout << "         keresztnev: "; cin >> knev0;
        p->DelTanulo(vnev0, knev0);
        p->KiirElore();
      }
    }
    if (p->GetLetszam())
    {
      p->Szamol();
      cout << endl;
      cout << "Az evfolyam atlaga : " << p->GetAtlag() << endl;
      cout << "Maximalis osztondij: " << p->GetMaxOsztondij()<< endl;
      cout << "Letszam            : " << p->GetLetszam() << endl;
    }
    else
      cout<<"Ures a lista";
  }
  delete p;
  cin.get(); cin.get();
  return 0;
}
```

A hasonló listakezelő feladatok megoldása során a nehézséget a mutatók szakszerű alkalmazása jelenti. A fenti megoldás során a **char *** mutatókkal azonosított karaktersorozatok helyett a ***string*** típust használtuk. A következőkben a szabványos sablontár újabb elemét hívjuk segítségül, a dinamikus vektorok osztályát (***vector***).

A *Tan_02_String* feladat megoldásában a lista szervezéséhez használjunk STL tároló objektumot! (*Tan_02_Vector*)

A szabványos sablontár (STL) osztályai közül a legegyszerűbbet választjuk ki a lista szervezésére, a dinamikus vektorok osztályát (***vector***), melynek példányát (*dv*) tagként helyezzük el a *Lista* osztályban. Mivel a ***vector*** egy általánosított osztály, a példányosítás során meg kell adnunk az elemek típusát (*Tanulo*).

```
class Lista
{
  private:
    vector<Tanulo> dv;
    Tanulo *elso;
    Tanulo *utolso;
    int MaxOsztondij;
    double TanulokAtlaga;
    int Letszam;
  public:
    enum {Elore, Hatra};
    Lista();
    ~Lista();
    void AddTanulo(const string & vnev0, const string & knev0,
                   double atlag0, int osztondij0, int irany);
    void KiirElore();
    void Szamol();
    double GetAtlag(){ return TanulokAtlaga;}
    int GetMaxOsztondij() {return MaxOsztondij;}
    void DelTanulo(const string & vnev0, const string & knev0);
    int GetLetszam(){return Letszam; }
};
```

Mint láthatjuk, a *Lista* osztály felülete változatlan maradt, csak a tagfüggvények tartalmát cseréljük le.

További előnye ennek a megoldásnak, hogy a listaműveletekkel kizárólag a *Lista* osztály foglalkozik, így a *Tanulo* osztály csak a tanulók adatait kezeli, kiegészítve a megjelenítés lehetőségével:

```
class Tanulo
{
  public:
    string vnev;
    string knev;
    double atlag;
    int osztondij;
  public:
    Tanulo(const string & vnev0, const string & knev0,
           double atlag0, int osztondij0);
    ~Tanulo() {}
    friend ostream & operator<<(ostream & os, const Tanulo & t);
};

// Paraméterezett konstruktor
Tanulo::Tanulo(const string & vnev0, const string & knev0,
               double atlag0, int osztondij0)
{
  vnev = vnev0;
  knev = knev0;
  atlag = atlag0;
  osztondij = osztondij0;
}

// Kiíró operátor
ostream & operator<<(ostream & os, const Tanulo & t)
{
   cout << "Nev        : " << t.vnev << "  " << t.knev << endl;
   cout << "  atlag    : " << t.atlag << endl;
   cout << "  osztondij: " << t.osztondij << endl;
   return os;
}
```

A *Lista* osztály tagfüggvényeinek tartalma jelentősen megváltozott. A szokásos mutatókkal végzett lépéseket, STL-elemek váltották fel, így például a konstruktor leegyszerűsödött, a destruktor pedig üressé vált.

```
Lista::Lista()
{
  TanulokAtlaga = 0;
  MaxOsztondij = 0;
  Letszam = 0;
}
```

A újfajta listakezelés során a *Lista* osztály *vector<Tanulo>* típusú *dv* adattagját használjuk, a tárolt adatok eléréséhez pedig ún. általánosított mutatókat, iterátorokat alkalmazunk. A ***begin()*** iterátor a lista elejére, az ***end()*** pedig a lista végére mutat, az elemek beszúrását az ***insert()***, törlését pedig az ***erase()*** tagfüggvénnyel végezzük.

```
void Lista::AddTanulo(const string & vnev0, const string & knev0,
                      double atlag0, int osztondij0, int irany)
{
  Letszam++;                          // létszám növelése
  Tanulo t(vnev0, knev0,atlag0,osztondij0);
  if (irany == Lista::Elore)
    dv.insert(dv.begin(), t);        // első elé fűz
  else
    dv.insert(dv.end(), t);          // az utolsó mögé fűz
}
```

A teljes lista bejárása esetén szükség van egy külső iterátorra, melyet a mutatókhoz hasonlóan léptethetünk a következő elemre, illetve mellyel elérhetjük az aktuális listaelemet.

```
void Lista::Szamol()
{ vector<Tanulo>::const_iterator p;    // iterátor a lista bejárásához
  int db = 0;
  for (p= dv.begin(); p!=dv.end(); p++) // az első elemtől az utolsóig
  {
    TanulokAtlaga += p->atlag;
    db++;
    if (MaxOsztondij < p->osztondij)
       MaxOsztondij = p->osztondij;
  }
  TanulokAtlaga /= db;
}
```

Külön iterátorral (*out*) gondoskodhatunk az kiírás automatikus elvégzéséről. Ebben az esetben a fenti **for** ciklus helyett elegendő az elemeket erre az kiviteli iterátorra rámásolnunk.

```
void Lista::KiirElore()
{  // ostream iterator a kiirashoz
   ostream_iterator<Tanulo>out(cout,"\n");
   copy(dv.begin(), dv.end(), out);    // a kiirás az elejétől a végéig
}
```

A törlendő elem megkereséséhez szintén végiglépkedünk a listán, ellenőrizve a nevek azonosságát. Amennyiben a megoldásból eltávolítjuk a **break** utasítást, nemcsak az első megtalált elemet távolíthatjuk el a listából, hanem az összeset.

```
void Lista::DelTanulo(const string & vnev0, const string & knev0)
{
  vector<Tanulo>::iterator p;   // iterátor a lista bejárásához
  for (p= dv.begin(); p!=dv.end(); p++) // az első elemtől az utolsóig
    if (p->vnev == vnev0 && p->knev == knev0 )
    {
      Letszam--;
      dv.erase(p);
      break; // a break nélkül minden adott nevű elemet töröl
      p--;
    }
}
```

Mivel a *Lista* osztály felhasználói felülete változatlan maradt, a futtatáshoz használt főfüggvényben sem kellett semmit módosítanunk. (A ***main()*** függvény kódja az előző feladat megoldásánál megtekinthető.)

19.4 A fastruktúra (tree) megvalósítása

Készítsünk fastruktúrán alapuló, egész számokat növekvő sorrendbe rendező programot! (*Tree1*)

A fastruktúra csomópontokból és a csomópontok közötti kapcsolatokból épül fel. Esetünkben minden csomópontból legfeljebb 2 kapcsolat indul ki (bal és jobb). A megoldást mégsem bináris fa, hiszen az egy kapcsolatot is megengedjük, azonban kis módosítással azzá tehető. A kapcsolat-információk mellett a csomópontok tartalmazzák még a csomóponti adatot (*Info*), melynek típusát – az egyszerűség kedvéért – típusdefinícióval adtuk meg a *TREE.H* deklarációs állományban:

```
typedef int Adatelem;
```

Ugyancsak itt találjuk a csomópontot leíró osztály deklarációját is:

```
class Csomopont
{
  friend class Tree;
public:
  Csomopont(const Adatelem &);    // konstruktor
  virtual Adatelem Feldolgoz();   // adatfeldolgozó tagfüggvény
protected:
  Csomopont *bal;                 // bal oldali kapcsolat
  Adatelem Info;                  // adat
  Csomopont *jobb;                // jobb oldali kapcsolat
};
```

A *Csomopont* osztály tagfüggvényeit a *TREE.CPP* fájl tartalmazza.

```
Csomopont::Csomopont(const Adatelem &adat)  // konstruktor
{
  Info = adat;
  bal = jobb = NULL;
}

Adatelem Csomopont::Feldolgoz()
{
  cout << setw(4) << Info;
  return Info;
}
```

A *Feldolgoz()* tagfüggvény segítségével a csomópontokon elvégzendő műveleteket definiálhatunk. A példában csak megjelenítjük a tárolt adatot, illetve visszaadjuk függvényértékként.

Következő lépésben a kapcsolatok kezeléséről kell gondoskodnunk. A műveleteket a *Tree* nevű osztályban gyűjtöttük csokorba:

```
class Tree
{
public:
  Tree();
  ~Tree();
  void Beilleszt(const Adatelem &); // új elem felvitele
  void Torol(const Adatelem &);     // elem törlése
  void Lebont();                    // a fa megsemmisítése
  void Megjelenit();                // a fa sematikus megjelenítése

  // Kivülről is elérhető bejárási függvények
  void Elozetes();                 // feldolgoz, bal, jobb
  void Utolagos();                 // bal, jobb, feldolgoz
  void Kozbeni();                  // bal, feldolgoz, jobb

private:
  int level;            // a szintet jelölő segédváltozó
  Csomopont * root;     // a fa gyökere
  // új csomópont felvitele a fastruktúrába
  void Beilleszt_Csomopont(Csomopont  * & , const Adatelem &);
  // elemmel kijelölt ág törlése
  void Torol_Csomopont(Csomopont * &, const  Adatelem & );
  // a teljes fa megsemmisítése
  void Lebont_Csomopont(Csomopont * & );
  // a teljes fa megjelenítése
  void Megjelenit_Csomopont(Csomopont * & p);
  // a tagfüggvényekből használható bejárási függvények
  void Elozetes_Csomopont(Csomopont *);
  void Utolagos_Csomopont(Csomopont *);
  void Kozbeni_Csomopont(Csomopont *);
};
```

Az osztály *root* adattagja a fastruktúra gyökerére mutató pointer, míg a *level* adattagot a szemléletes megjelenítés során használjuk.

A *Tree* osztály definícióját szintén a *TREE.CPP* állományban találjuk, az alábbiakban csak a legfontosabb műveletek bemutatására szorítkozunk. A legtöbb művelethez két tagfüggvény is tartozik. Az egyik nyilvános elérésű és legfeljebb az *Adatelem* típusú paraméterrel rendelkezik, mint például az ág törlése művelet:

```
void Tree::Torol(const Adatelem & elem)
{
  Torol_Csomopont(root, elem);
}
```

Ebből hívunk egy belső tagfüggvényt, amely a gyökértől kiindulva rekurzív módon bejárja a fát, és elvégzi a csomópontok megsemmisítését. Talán nem a leghatékonyabb, azonban biztos, hogy a leginkább áttekinthető megoldást ad a rekurzív függvények használata.

```
void Tree::Torol_Csomopont(Csomopont * & p, const Adatelem & elem)
{
  if (p != NULL)
  {
    if (p->Info==elem)
    {
       Lebont_Csomopont(p->bal);
       Lebont_Csomopont(p->jobb);
       delete p;
       p=NULL;
       return;
    }
    Torol_Csomopont(p->bal, elem);
    Torol_Csomopont(p->jobb, elem);
  }
}
```

Az elmondottak alapján készültek a teljes fát megsemmisítő *Destroy()*, és a fa különböző bejárását biztosító bejárási függvények (*Elozetes()*, *Utolagos()*, *Kozbeni()*).

Valamelyest eltérő megoldást alkalmaztunk az új elem felvitelét végző *Beilleszt()* tagfüggvény esetében. Mivel a beillesztés az elemek növekvő sorrendjében történik, fontos, hogy az *Adatelem* típushoz legyenek definiálva összehasonlító műveletek (<, >, ==).

```
void Tree::Beilleszt(const Adatelem & elem)
{
  Beilleszt_Csomopont(root, elem);
}
```

```
void Tree::Beilleszt_Csomopont(Csomopont * & p,
                                const Adatelem & elem)
{ if (p == NULL)  // üres csomópont - új csomópont felvétele
    {
      p = new Csomopont(elem);
      p->bal = p->jobb = NULL;
    }
  else
    { if (p->Info > elem )
      Beilleszt_Csomopont( p->bal, elem);
    else // az elem nagyobb vagy egyenlő, mint a tartalmak
      Beilleszt_Csomopont(p->jobb, elem);
    }
}
```

Az elmondottak alapján már érthető a fastruktúra működését bemutató főfüggvény, amelyben egész számokat helyezünk el a fastruktúra csomópontjaiban:

```
#include <iostream>
#include "Tree.h"
using namespace std;

main()
{ Tree fa;
  int e, n;
  cout << "Adatok szama: "; cin >> n;
  if ( n > 0)
    for (int i = 0; i< n ; i++)
    { cin>>e;
      fa.Beilleszt(e);
    }

  cin.get();
  cout << endl <<"Elozetes bejaras:" << endl;
  fa.Elozetes();
  cout << endl <<"Kozbeni bejaras:" << endl;
  fa.Kozbeni();
  cout << endl <<"Utolagos bejaras:" << endl;
  fa.Utolagos();
  cout<<endl;
  cin.get();
  cout<<"A fa felepitese:"<<endl;
  fa.Megjelenit();
  cin.get();
  cout<<"A torolni kivant agat azonosito csomponti adat:"<<endl;
  cin>>e;
  fa.Torol(e);
  cout<<endl;
  cout << endl <<"Kozbeni bejaras:" << endl;
  fa.Kozbeni();
  cout << endl << "Lebontas" << endl;
  fa.Lebont();
  cin.get();
  cin.get();
}
```

A program futásának eredménye:

```
Adatok szama: 5
79
12
23
7
35

Elozetes bejaras:
  79  12   7  23  35
Kozbeni bejaras:
   7  12  23  35  79
Utolagos bejaras:
   7  35  23  12  79

A fa felepitese:
79
..12
....7
....23
......35

A torolni kivant agat azonosito csomponti adat:
7

Kozbeni bejaras:
  12  23  35  79
Lebontas
```

Az előző alfejezet (19.3) *Tanulo* osztályának felhasználásával készítsünk fastruktúrán alapuló, névsor szerinti rendezést végző programot! (*Tree2*)

Első lépésben a *Tanulo* osztályt ellátjuk azokkal a műveletekkel (összehasonlítás, I/O), melyek megléte szükséges a névsor szerinti rendezés programozásához. Az elkészült operátorfüggvények nagyon egyszerűek így nem magyarázzuk azok működését. Az osztály definícióját a *TANULO.H* és a *TANULO.CPP* állományok tartalmazzák.

```
//TANULO.H
#include <iostream>
#include <string>
using namespace std;

class Tanulo
{
  public:
    string vnev;
    string knev;
    double atlag;
    int osztondij;
  public:
    Tanulo(const string & vnev0, const string & knev0,
           double atlag0, int osztondij0);
    Tanulo() {}
    ~Tanulo() {}
    friend ostream & operator<<(ostream & os, const Tanulo & t);
    friend istream & operator>>(istream & is, Tanulo & t);
    bool operator  < (const Tanulo & t);
    bool operator  > ( const Tanulo & t);
    bool operator  == (const Tanulo & t);
};

//TANULO.CPP
#include "Tanulo.h"

Tanulo::Tanulo(const string & vnev0, const string & knev0,
               double atlag0, int osztondij0)
{
  vnev=vnev0; knev=knev0;
  atlag = atlag0; osztondij = osztondij0;
}

bool Tanulo::operator < (const Tanulo & t)
{
   return vnev+knev < t.vnev+t.knev;
}

bool Tanulo::operator == (const Tanulo & t)
{
   return vnev+knev == t.vnev+t.knev;
}
bool Tanulo::operator > (const Tanulo & t)
{
   return vnev+knev > t.vnev+t.knev;
}

ostream & operator<<(ostream & os, const Tanulo & t)
{
   cout << "Nev : " << t.vnev << "  " << t.knev ;
   cout << "\tatlag : " << t.atlag ;
   cout << "\tosztondij: " << t.osztondij << endl;
   return os;
}

istream & operator>>(istream & is, Tanulo & t)
{     cout << "Vezeteknev: ";  cin >> t.vnev;
      cout << "Keresztnev: ";  cin >> t.knev;
      cout << "Atlag     : ";  cin >> t.atlag;
      cout << "Osztondij : ";  cin >> t.osztondij;
      cout << endl;
   return is;
}
```

A *TREE.H* és *MAIN.CPP* állományokban végzett módosítások után az alkalmazás azonnal futtatható az új adattípussal:

```
#include "Tanulo.h"                 // a TREE.H-ban
typedef Tanulo Adatelem;

Tanulo e;                           // a MAIN.CPP-ben
```

A program futásának eredménye:

```
Adatok szama: 3
Vezeteknev: Kiss
Keresztnev: Laszlo
Atlag     : 3.5
Osztondij : 3000

Vezeteknev: Nagy
Keresztnev: Zoltan
Atlag     : 4
Osztondij : 4200

Vezeteknev: Fazekas
Keresztnev: Istvan
Atlag     : 4.5
Osztondij : 5000

Elozetes bejaras:
Nev : Kiss  Laszlo      atlag : 3.5     osztondij: 3000
Nev : Fazekas  Istvan   atlag : 4.5     osztondij: 5000
Nev : Nagy  Zoltan      atlag : 4       osztondij: 4200

Kozbeni bejaras:
Nev : Fazekas  Istvan   atlag : 4.5     osztondij: 5000
Nev : Kiss  Laszlo      atlag : 3.5     osztondij: 3000
Nev : Nagy  Zoltan      atlag : 4       osztondij: 4200

Utolagos bejaras:
Nev : Fazekas  Istvan   atlag : 4.5     osztondij: 5000
Nev : Nagy  Zoltan      atlag : 4       osztondij: 4200
Nev : Kiss  Laszlo      atlag : 3.5     osztondij: 3000


A fa felepitese:
Nev : Kiss  Laszlo      atlag : 3.5     osztondij: 3000

..Nev : Fazekas  Istvan atlag : 4.5     osztondij: 5000

..Nev : Nagy  Zoltan    atlag : 4       osztondij: 4200

A torolni kivant agat azonosito csomponti adat:
Vezeteknev: Fazekas
Keresztnev: Istvan
Atlag     : 4.5
Osztondij : 5000

Kozbeni bejaras:
Nev : Kiss  Laszlo      atlag : 3.5     osztondij: 3000
Nev : Nagy  Zoltan      atlag : 4       osztondij: 4200

Lebontas
```

19.5 Kettős lista használata

Készítsünk a tanulók adatait kettős listában tároló, illetve kezelő programot! (*DLista*)

A feladat többféleképpen is megoldható, azonban mi külön célként tűztük ki a saját osztálysablon felhasználását. Ennek megfelelően először elkészítettük a lineáris, rendezett lista működését megvalósító osztályt (*Lista*), és elmentettük a *LISTA.H* állományban. (Mivel ez a teljes osztálydefiníciót tárolja, csak egyszer szabad beépítenünk a projektbe.) A listasablon tetszőleges adattípussal specializálható, amely rendelkezik a ==, a <= és a > relációs, illetve az adatkiviteli << operátorok átdefiniált változataival.

A megoldásban a kettős lista kialakítása érdekében kétszer specializáltuk a listasablont: a *Tanulo* osztállyal, illetve a *Tanulo* osztály tagjaként megadott jegyek lista létrehozásához az **int** típussal.

A *Tanulo* osztályt kicsit leegyszerűsítettük, csak a feladat megoldása szempontjából lényeges tagokat hagytuk meg. Az osztály deklarációja a *TANULO.H* állományban (itt építjük be a projektbe a *LISTA.H* sablonfájlt):

```
//TANULO.H
#include <iostream>
#include <string>
#include <cstdlib>
#include "Lista.h"
using namespace std;
class Tanulo
{
  public:
    string vnev;
    string knev;
    Lista <int> jegyek;  // specializáció
  public:
    Tanulo(const string & vnev0, const string & knev0);
    double Atlag();
    Tanulo() {}
    ~Tanulo() {}
    friend ostream & operator<<(ostream & os, const Tanulo & t);
    bool operator  > ( const Tanulo & t);
    bool operator  <= (const Tanulo & t);
    bool operator  == (const Tanulo & t);
};
```

Az osztály tagfüggvényei a *TANULO.CPP* fájlban találhatók:

```
#include "Tanulo.h"

Tanulo::Tanulo(const string & vnev0, const string & knev0)
{  vnev=vnev0; knev=knev0; }
```

```
bool Tanulo::operator == (const Tanulo & t)
{
   return vnev+knev == t.vnev+t.knev;
}

bool Tanulo::operator <= (const Tanulo & t)
{
   return vnev+knev <= t.vnev+t.knev;
}

bool Tanulo::operator > (const Tanulo & t)
{
   return vnev+knev > t.vnev+t.knev;
}
```

A kiírás során a név mellett az osztályzatok és az átlag is megjelenik. Mivel a saját tagfüggvényeinknél általában konstans referenciával adjuk át az objektumpéldányokat, vannak esetek, amikor típusmódosítóval le kell „szednünk" a hivatkozásról az állandó jellemzőt. Erre a célra a C++ nyelv **const_cast<>()** típusátalakító műveletét használjuk:

```
ostream & operator<<(ostream & os, const Tanulo & t)
{
   cout<< "\nNev     : " << t.vnev << "  " << t.knev;
   cout<<"\nJegyek   : "; const_cast< Lista<int>& >
                                    (t.jegyek).OutIterator();
   cout<< "\nAtlag   : "<< const_cast<Tanulo &>(t).Atlag()<<endl;
   return os;
}
```

A tárolt osztályzatok átlagát függvényhívással határozzuk meg, felhasználva a lista elemeinek számát (*Darab()*), illetve a listaelemekben tárolt adatokat, amit a lista iterátorától kapunk meg.

```
double Tanulo::Atlag()
{
   double atlag=0;
   int n = jegyek.Darab();
   for (int i=0; i<n; i++)
     atlag+=jegyek();
   return atlag/n;
}
```

Nézzük meg a listakezelés osztályának teljes definícióját! Ne feledjük el, hogy ez osztálysablon, melynek általánosított típusa az *Adat*! Az osztálysablont a *LISTA.H* állomány tartalmazza:

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
template <class Adat> class Lista
{
 private:
  struct ListaElem
  {
    Adat Info;
    ListaElem *Link;
  };
  int elemszam;
  ListaElem *bptr; // a lista eleje
  ListaElem *iptr; // az iteratornak
 public:
  Lista(const Adat & data);
  Lista() { bptr  = iptr = 0;  elemszam=0; }
  ~Lista() ;

  void OutIterator();
  Adat& operator()();
  void Beszur(const Adat &);
  void Torol(const Adat &);
  bool Keres(const Adat &);
  int Darab() {return elemszam;}
};
```

A paraméterezett konstruktorral egy adott elemmel indíthatjuk a listát, míg az alapértelmezett konstruktor üres listát inicializál:

```
template <class Adat> Lista<Adat>::Lista(const Adat & adat)
{
  ListaElem * newptr;
  newptr = new ListaElem;
  newptr->Info = adat;
  newptr->Link = NULL;
  iptr = bptr = newptr;
  elemszam=1;
}
```

A destruktor megszünteti a lista dinamikus helyfoglalású elemeit:

```
template <class Adat> Lista<Adat>::~Lista()
{
   ListaElem *rptr, *tptr;
   for(tptr = bptr; tptr != NULL; tptr = rptr)
   {
     rptr = tptr->Link;
     tptr->Link = NULL;
     delete tptr;
   }
  bptr=NULL;
  elemszam=0;
}
```

Az egyik legfontosabb listaművelet az új elem felvitele. Mivel az elemek nagyság szerint rendezetten kerülnek be a listába, az elem beillesztését egy sor vizsgálat előzi meg.

```
template <class Adat> void Lista<Adat>::Beszur(const Adat & adat)
{
    ListaElem * sptr, *tptr, *newptr;
    // helyfoglalás az új elem számára
    newptr = new ListaElem;
    newptr->Info= adat;
    elemszam++;
    if (bptr==NULL)             // első elem
    {
       bptr=newptr;
       bptr->Link=NULL;
       return;
    }
    // ha az új adat <= az első elemnél, akkor beszúrjuk elé
    if (const_cast<Adat & >(adat) <= bptr->Info)
    {
       newptr->Link=bptr;
       bptr=newptr;
       return;
     }
    // Megkeressük, hogy melyik két elem közé esik az adat
    // ellenőrizzük a nála nagyobbat, vagy a lista végét
    sptr=bptr;              // első
    tptr=bptr->Link;        // második
    while (tptr != NULL && tptr->Info<=adat)
    {
      sptr=tptr;            // továbblép
      tptr=tptr->Link;
    }
   // mivel nincs az adat-nál nagyobb elem, a lista végére tesszük
   if (tptr==NULL && sptr->Info<=adat)
   {
      sptr->Link=newptr;
      newptr->Link=NULL;
      return;
   }
   // ha az utolsó elem nagyobb az adat-nál
   if (tptr==NULL && sptr->Info>adat)
   { tptr=sptr;
     sptr=bptr;
     // megkeressük az utolsó előttit
     while (sptr->Link!=tptr)
        sptr=sptr->Link;
   }
   // az *sptr és a *tptr közé szúrjuk be az új elemet
   newptr->Link=tptr;
   sptr->Link=newptr;
}
```

A listából való törlés feltétele, hogy a listaelemben tárolt adat megegyezzen az argumentumként átadottal. Itt is külön vizsgálatokat igényel az első elem törlése.

```
template <class Adat> void Lista<Adat>::Torol(const Adat & adat)
{
   ListaElem *rptr, *sptr;
  rptr=bptr;
  bool talalt=false;
  while (rptr!=NULL)
  { if (rptr->Info == adat)
    {
      talalt=true;
      break;
    }
     rptr = rptr->Link;
  }
  if (talalt) {
    if (rptr==bptr) // első elem
    {
       bptr=rptr->Link;
       delete rptr;
       elemszam--;
    }
    else
    {  // ha nem első megkeressük az előtte álló elemet
       sptr=bptr;
       while (sptr->Link!=rptr)
          sptr=sptr->Link;
       sptr->Link=rptr->Link;
       delete rptr;
       elemszam--;
    }
  }
}
```

Az alábbi *Keres()* tagfüggvény arról ad felvilágosítást, hogy egy adat benne van-e a listában, vagy sem.

```
template <class Adat> bool Lista<Adat>::Keres(const Adat & adat)
{
  ListaElem *rptr;
  rptr=bptr;
  bool talalt = false;
  while (rptr!=NULL)
  {
    if (rptr->Info == adat)
    {
      talalt = true;
      break;
    }
    rptr = rptr->Link;
  }
  return talalt;
 }
```

A *Lista* osztályhoz kétféle bejárási megoldást is készítettünk. Az *OutIterator()* tagfüggvény végigmegy a listán, és megjeleníti a tárolt adatot a szabványos kimeneten. A függvényhívás operátorának túlterhelt változata minden híváskor a következő listaelem adatrészére hivatkozó referenciával tér vissza.

```
template <class Adat> void Lista<Adat>::OutIterator()
{
  ListaElem * rptr;
  rptr = bptr;
  while (rptr != NULL)
  {
      cout << setw(5) << rptr->Info;
      rptr = rptr ->Link;
  }
}
template <class Adat> Adat & Lista<Adat>::operator()()
{
  ListaElem * sptr;
  if (iptr!=0)
     sptr=iptr;
  else
     sptr=iptr=bptr;
  iptr=iptr->Link;
  return sptr->Info;
}
```

A kettős lista működését egy egyszerű főprogramban követhetjük nyomon (*MAIN.CPP*). Az adatok egy részét konstans, más részét pedig véletlenszerű értékekkel adjuk meg.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

#include "Tanulo.h"
int main()
{
  srand((unsigned)time(NULL));
  Lista<Tanulo> Tankor;

  Tanulo h("Kiss", "Aladar");
  for(int i=1; i<rand()%5+3; i++)
    h.jegyek.Beszur(rand()%5+1);
  Tankor.Beszur(h);

  h=Tanulo("Adam","Eva");
  for (int i=1; i<rand()%5+3; i++)
    h.jegyek.Beszur(rand()%5+1);
  Tankor.Beszur(h);

  h=Tanulo("Zalai","Norbert");
  for (int i=1; i<rand()%5+3; i++)
    h.jegyek.Beszur(rand()%5+1);
  Tankor.Beszur(h);
  Tankor.OutIterator();

  h=Tanulo ("Kiss", "Aladar");
  Tankor.Torol(h);
  Tankor.OutIterator();
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Nev    : Adam  Eva
Jegyek :     1    3    4    5
Atlag  : 3.25


Nev    : Kiss  Aladar
Jegyek :     1    1    5    5
Atlag  : 3

Nev    : Zalai  Norbert
Jegyek :     1    3
Atlag  : 2

Nev    : Adam  Eva
Jegyek :     1    3    4    5
Atlag  : 3.25

Nev    : Zalai  Norbert
Jegyek :     1    3
Atlag  : 2
```

A *Lista* sablon felhasználásával készítsünk dinamikus helyfoglalású, lottószámokat tároló listatömböt! (*ListaTomb*)

A feladat megoldásának lépései csupán formájukban térnek el a szokásos dinamikus memória-használattól. Az alábbi ***main()*** függvényben 12 szelvény kitöltéséhez generálunk lottószámokat. A megoldásban minden szelvényt egy-egy lista tárol, melyek objektumát szintén dinamikusan hozzuk létre az alábbiak szerint

```
Lista<int> *Lotto;
Lotto = new Lista<int>[12];
```

Minden egyes tömbelemlista létrejötte után azonnal meghívódik az alapértelmezés szerinti kontsruktor, így a listák üres listaként születnek meg. Felhívjuk a figyelmet a **delete** operátor helyes alkalmazására:

```
delete [] Lotto;
```

Amennyiben lefelejtjük a szögletes zárójelet, csak a *Lotto[0]* lista destruktora hívódik meg, így a többi 11 elemhez tartozó láncot nem szabadítjuk fel. E rövid megjegyzések után tekintsük a főmodult és a futási eredményeket!

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
#include "Lista.h"

int main()
{
   srand((unsigned)time(NULL));

   Lista<int> *Lotto;
   Lotto = new Lista<int>[12];

   int szam;
   for (int i=0; i<12; i++)
   {
      while (Lotto[i].Darab()<5)          // öt számot
      {
          szam=rand()%90+1;               // 90-ből
          if (!Lotto[i].Keres(szam))
             Lotto[i].Beszur(szam);
      }
      Lotto[i].OutIterator();
      cout<<endl;
   }
   delete [] Lotto;
   cin.get();
   return 0;
}
```

A program futásának eredménye:

```
   22   32   51   52   61
   16   36   47   61   80
    4    7    9   29   89
   42   43   60   80   85
    1    6    8   60   74
    2   15   17   61   77
    9   23   63   77   83
   15   23   24   68   82
    5   23   40   45   82
    2   20   35   64   88
    8   12   20   43   60
   52   68   74   80   90
```

Függelék

F1. A Dev-C++ fejlesztői környezet használata

F1.1 Objektum-orientált program állományai

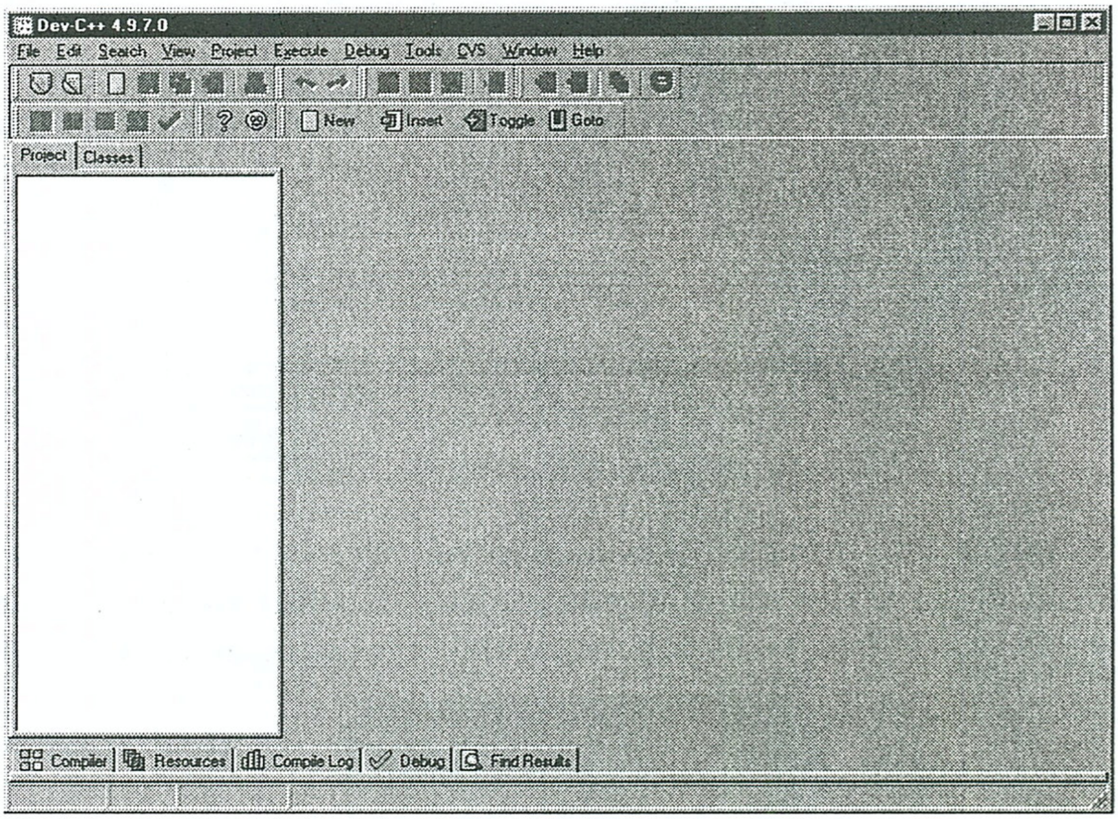
Az objektum-orientált program több állományból épül fel, a feladatok megoldása során általában az alábbiakat hoztuk létre:

- A *main()* függvényt tartalmazó *MAIN.CPP* állomány.
- Az osztály(ok) deklarációját tartalmazó *OSZTALY.H* állomány.
- Az osztály(ok) tagfüggvényeinek definícióját tartalmazó *OSZTALY.CPP* fájl.
- A program állományait a *PROJECT1.DEV* fogja össze.
- A futtatható program neve azonos a projekt nevével, például *PROJECT1.EXE.*

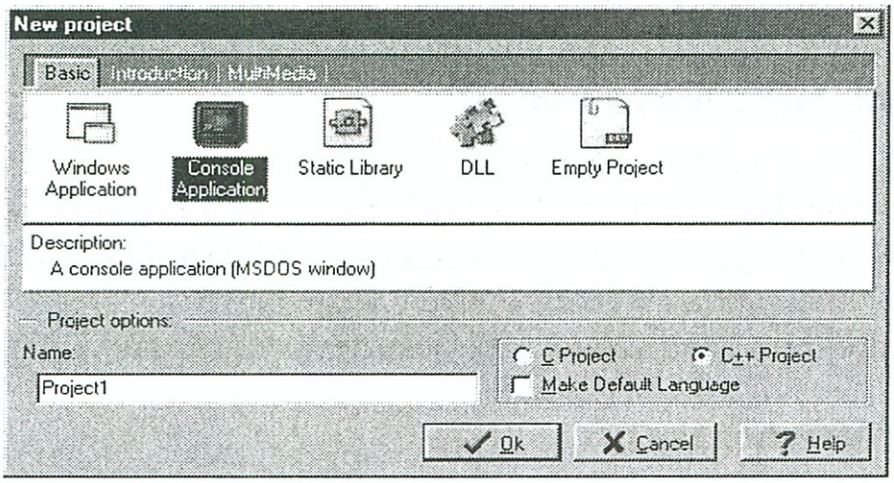
F1.2 Új project létrehozása

Először telepítsük a CD-mellékleten található 4.9.7.0 verziójú, szabad felhasználású fordítót, Dev-C++ és tegyük ki az Asztalra az indító ikonját:

A *devcpp* ikonon kattintva a fejlesztői környezet ablaka jelenik meg.


Dev-C++ 4.9.7.0
File Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Project Classes
Compiler Resources Compile Log Debug Find Results

A **File/New/Project...** menüpont kiválasztásával új projektet hozunk létre, ha a megjelenő párbeszédablakban kijelöljük a *Console Application* ikont és kattintunk az *OK* nyomógombra.


New project
Basic
Introduction
MultiMedia
Windows Application
Console Application
Static Library
DLL
Empty Project
Project options:
Ok
Cancel
Help

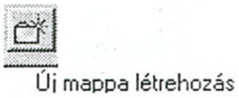
Description:
A console application (MSDOS window)

Name: Project1

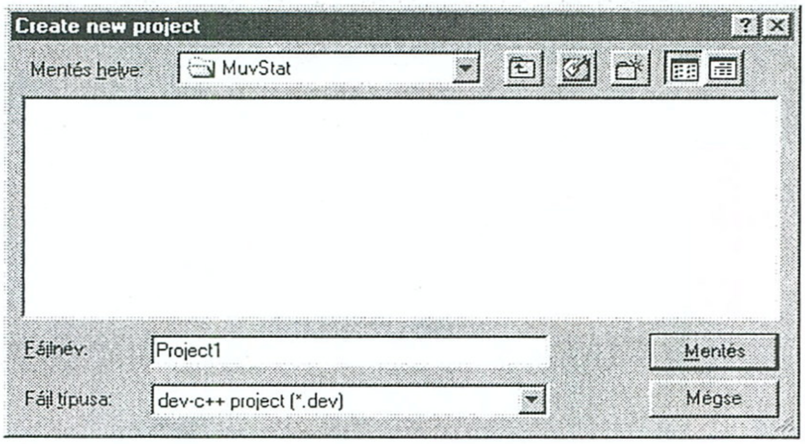
○ C Project ● C++ Project

☐ Make Default Language

Példaként bemutatjuk a *MuvStat* feladat projektjének felépítését. A *Project1* állomány számára *MuvStat* néven hozzunk létre egy könyvtárat az *Új mappa létrehozása* ikonon kattintva.


Új mappa létrehozás

A létrehozott könyvtár nevén kattintva belépünk a könyvtárba, és a felajánlott *Project1* nevet elfogadva, megnyomjuk a *Mentés* nyomógombot.

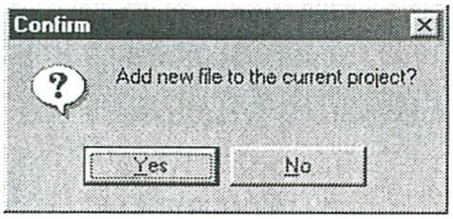

Create new project
Mentés
Mégse

Mentés helye: MuvStat

Fájlnév: Project1

Fájl típusa: dev-c++ project (*.dev)

A **File/New/Source File** menüpont kiválasztásával vagy *<Ctrl+N>* gyorsítóbillentyű megnyomásával új forrásállományt hozhatunk létre és adhatunk az aktuális *projekthez*, ha a megjelenő párbeszédablakban megnyomjuk a *Yes* nyomógombot.


Confirm
Yes
No

Add new file to the current project?

A forrásállomány *Untitled1* néven jön létre, melyet a **File/Save As...** menüpont kiválasztásával *OSZTALY.H* állománynak nevezzünk el, majd hozzunk létre egy újabb forrásállományt *OSZTALY.CPP* néven.

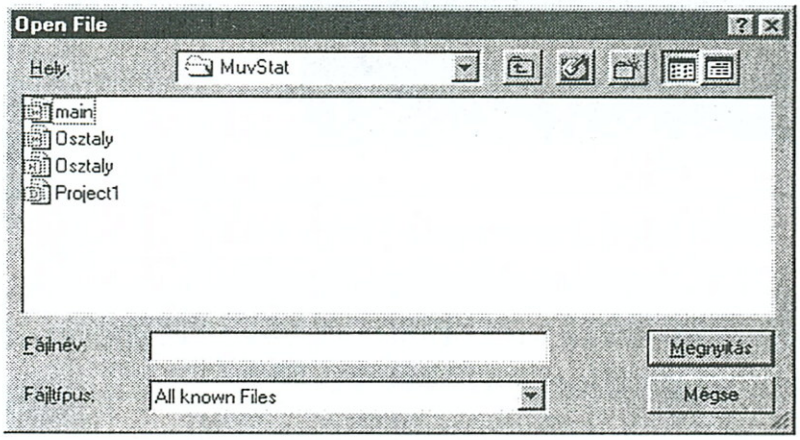
A ***main()*** függvény automatikusan létrejön, melyet a **File/Save As...** menüpont kiválasztásával *MAIN.CPP* állománynak nevezzünk el.

A felajánlott programot módosítjuk a táblázat jobb oldala szerint:

```
#include <iostream>
#include <stdlib.h>

int main(int argc, char *argv[])
{

  system("PAUSE");
  return 0;
}
```

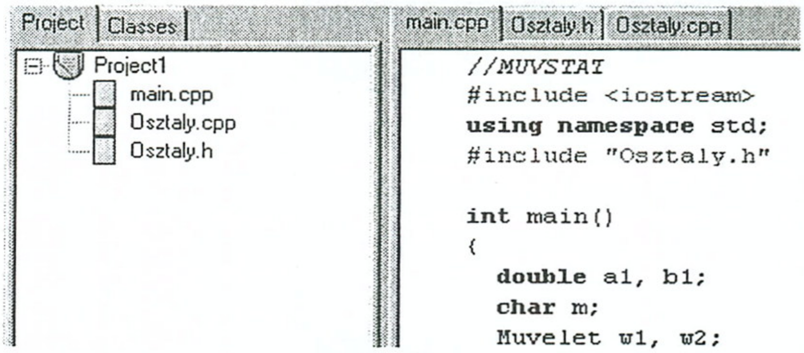
```
#include <iostream>
using namespace std;

int main()
{

  cin.get();
  return 0;

}
```

A *MuvStat* könyvtár szerkezete és tartalma:


Open File
Hely:
MuvStat
main
Osztaly
Osztaly
Project1
Fájlnév:
Fájltípus:
All known Files
Megnyitás
Mégse

A *MuvStat* könyvtár állományai:

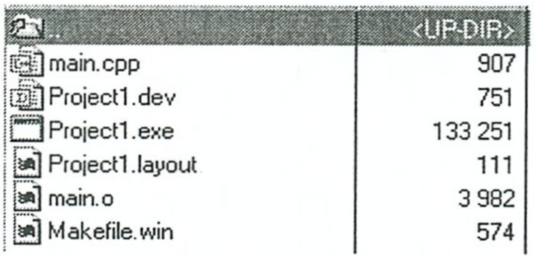
MAIN.CPP
OSZTALY.CPP
OSZTALY.H
PROJECT1.DEV


Project
Classes
Project1
main.cpp
Osztaly.cpp
Osztaly.h
main.cpp
Osztaly.h
Osztaly.cpp

```
//MUVSTAT
#include <iostream>
using namespace std;
#include "Osztaly.h"

int main()
{
  double a1, b1;
  char m;
  Muvelet w1, w2;
```

A bal oldali ablakban a *Project1* szerkezete látható.

A jobb oldali ablakban a *MAIN.CPP* állományban lévő ***main()*** függvény részlete.



..	<UP-DIR>
main.cpp	907
Project1.dev	751
Project1.exe	133 251
Project1.layout	111
main.o	3 982
Makefile.win	574

A lefordított program könyvtárában a *PROJECT1.EXE* a futtatható program. A program neve azonos a projekt nevével.

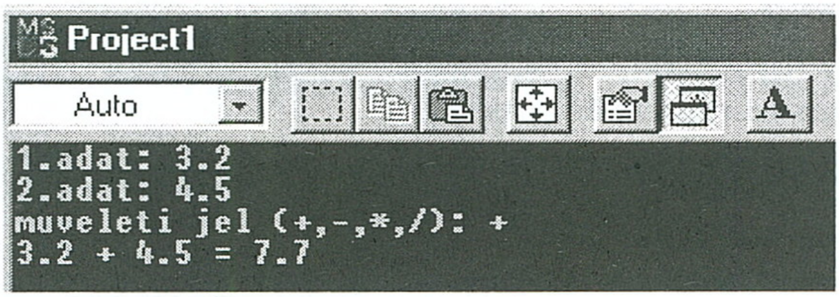
F1.3 A program futtatása

A programot az **Execute/Compile & Run** menüpont kiválasztásával vagy az *<F9>* gyorsítóbillentyű segítségével futtathatjuk.

Az eszköztár gombjaival a következő műveleteket végezhetjük el:

⊞	*Compile <Ctrl+F9>*	Fordítás.
☐	*Run <Ctrl+F10>*	Futtatás.
⊞	*Compile & Run <F9>*	Fordítás és futtatás.
⊞	*Rebuild All <Ctrl+F11>*	A program újrafordítása.

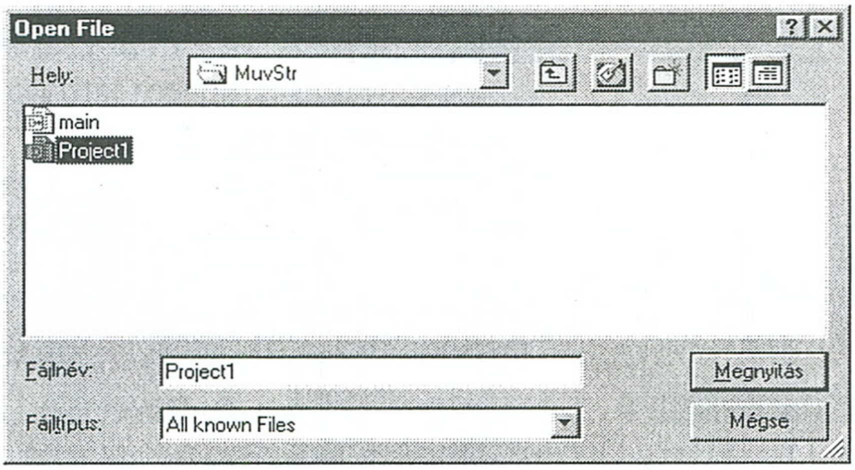
A program futásának eredménye a projekt nevének megfelelő fejlécű ablakban jelenik meg:



```
1.adat: 3.2
2.adat: 4.5
muveleti jel (+,-,*,/): +
3.2 + 4.5 = 7.7
```

F1.4 Project betöltése

A **File/Open Project or File...** menüpont kiválasztásával, vagy *<Ctrl+O>* gyorsítóbillentyű megnyomásával megkeressük a könyvtárat, kijelöljük a *Project1* fájlnevet és a *Megnyitás* nyomógombon kattintva a program állományai betöltődnek.



Hely: MuvStr

Fájlnév: Project1

Fájltípus: All known Files

F2. C++ óravázlatok

Az F2. függelékben olyan oktatási anyag található, amely gyakorlatonként csoportosítja a C++ nyelv témaköreit. Az oktatási anyag bővebb, magyarázatokkal ellátott feladatlapjai PDF-formátumban megtalálhatók a CD-melléklet *Tanari kezikonyv* nevű könyvtárában. A feladatsorok felölelik a könyv teljes tartalmát, azonban az alapkönyv feladataitól eltérő feladatokon keresztül mutatjuk be a C++ nyelv lehetőségeit, és lépésről-lépésre haladunk az ismeretek elsajátításában. A gyakorlatok elején röviden ismertetjük az alfejezet témakörét és a hozzá tartozó mintapéldákat. A mintapéldák kidolgozott megoldásának célja, hogy segítse a további feladatok gördülékenyebb megoldását.

Az óravázlatokhoz tartozó forráskódok a gyakorlati foglalkozás sorszámának megfelelően alkönyvtárakba kerültek. Például, az első gyakorlat programjainak forráskódját a *GYAK01_CPP* alkönyvtár tartalmazza. Mivel a C++ alkalmazások általában több állományból épülnek fel, minden projektet külön könyvtárban tárolunk, a gyakorlati foglalkozás és a feladat sorszámának megfelelően. Például, a *GYAK1_1* alkönyvtár az első gyakorlat 1. feladatának megoldását tartalmazza, míg a *MINTA1_01* mappában az 1. gyakorlat 1. mintaprogramjának forráskódja található.

Gyakorlat	A gyakorlat tematikája
1.	C++ nyelv alapelemei, jelkészlete. Konstansok, megjegyzések. Egyszerű változók definiálása. Operátorok és kifejezések. A C++ program szerkezete. Szabványos input a ***cin***, szabványos output a ***cout*** adatfolyam használata, *iostream*.
2.	Összehasonlító, logikai és léptető operátorok. Bit- és biteltoló műveletek. A **sizeof** operátor. A C++ nyelv utasításai. Az **if** feltételes utasítás. A **switch** többirányú elágaztatás. A **break** utasítás.
3.	Ciklusutasítások: **for**, **while**, **do-while**. Definíciók bevitele az utasításokba.
4.	Mutatók és referenciák. Többszörös indirektségű mutatók. Típuskonverziók. Futásidejű típusazonosítás. Tömbváltozók definiálása. A **new** és a **delete** operátor használata.
5.	A felhasználó által definiált adattípusok: a **typedef** és a **struct**. Dinamikus helyfoglalású struktúrák létrehozása és megszüntetése.
6.	Függvények deklarációja és definíciója. Az **inline** függvények. Referencia, tömb és struktúra típusú paraméterek alkalmazása. Függvényre mutató pointer átadása függvénynek.

Gyakorlat	**A gyakorlat tematikája** (folytatás)
7.	Alapértelmezés szerinti (*default*) és változó hosszúságú argumentumok. Rekurzív függvények használata. A függvénynevek átdefiniálása (**overloading**). A ***main***() függvény paraméterei és visszatérési értéke. Karaktersorozat-argumentumok.
8.	Fájlkezelés. Különféle típusú adatok (karakterek, számok, karaktersorozatok stb.) fájlba írása és állományból való olvasása. Az ***ios*** osztály használata. Bináris fájlok kezelése. Struktúra adatainak kiírása bináris állományba, illetve visszaolvasása bináris állományból.
9.	Az objektum-orientált programozás (OOP) alapelvei: egységbezárás, öröklés, sokalakúság. Statikus helyfoglalású objektumpéldányok. Információ-rejtés. A **this** mutató. Az osztályok **friend** (barát) mechanizmusa. Barátosztály alkalmazása.
10.	A konstruktorok és a destruktor szerepe. Alapértelmezett, paraméteres és másoló konstruktor. Dinamikus helyfoglalású objektumpéldányok létrehozása és megszüntetése (a **new** és **delete** operátorok alkalmazása objektumok esetén).
11.	Operátorok túlterhelése (*operator overloading*). Felhasználó által definiált típuskonverzió. Az osztályok bővítése input/output műveletekkel. A C++ nyelv bővítése saját típussal. A **new** és a **delete** műveletek újraértelmezése.
12.	Az öröklés mechanizmusa, származtatott és alaposztály, többszörös öröklés. **public**, **private** és **protected** öröklés Többszörös öröklés bemutatása családfán.
13.	Objektumok zártsága, korai kötés (*early binding*). Sokalakúság (*polymorphism*) – virtuális tagfüggvények, késői kötés (*late binding*). Absztrakt osztály, tiszta virtuális tagfüggvények és statikus adattagok használata
14.	Az objektum, mint adattag – beágyazás. Általánosított osztályok (**template**). Saját névterek kialakítása.
15.	Láncolt listák: egyirányú lista készítése, kétirányú lista megvalósítása. Listaműveletek.
16.	Kivételkezelés: a **try**, a **throw** és a **catch** kulcsszavak használata. A **new** művelet ***bad_alloc*** kivétele.

1. Gyakorlat

Rövid elméleti összefoglaló

- A C++ nyelv hatékony, általános célú programozási nyelv, amely hagyományos fejlesztőeszközként és objektum-orientált programozási nyelvként egyaránt használható. Erősen típusos nyelv. Újdonságai: kivételek (*exception*) kezelése, valamint a paraméterezett típusok, osztályok és függvények használata (*templates*, sablonok).
- **A C++ nyelv alapelemei:**
  Az ANSI C++ szabvány a nyelv alapelemeit (nevek, számok, karakterek) *token*nek nevezi. A C++ nyelv betűérzékeny, megkülönbözteti a kis- és a nagybetűket.
- **A nyelv jelkészlete**: Az angol ábécé kis- és nagybetűi s egyéb jelek (! " # % & ' ( ) * + , - /:; < = >? [\ ] ^ _ { | } ~) s a nem látható jelek (*whitespace*) (szóköz, vízszintes és függőleges tabulátor, új sor ('\n') stb.), valamint a UniCode karakterek, amelyek csak megjegyzésekben szerepelhetnek.
- **A C++ nyelv azonosítói**: A legtöbb fordító 32-karakteres neveket használ, az első karakter betű vagy _ (aláhúzásjel), melyet betű, szám, illetve aláhúzásjel követhet. A legtöbb fordító kibővíti a szabványos kulcsszavakat saját jelentéssel: __try, __property stb.
- **Az Ansi C++ nyelv kulcsszavait** nem lehet átdefiniálni, új jelentéssel ellátni.
  Konstansok: C++ nyelvben: karakteres, logikai, egész, felsorolt és lebegőpontos konstansokat használhatunk:
  - A C++ nyelv *logikai konstansai* **true** és **false** lehetnek.
  - *Egyetlen mutatókonstanssal* rendelkezik ez a 0 (nulla), melyet a NULL szimbólummal jelölünk.
  - *Egész konstansok*: 12, 012, 0x20 (decimális, oktális, hexadecimális),
    32U, 43L (U, u - **unsigned**, L, l - **long**)
  - *Felsorolt konstansok*: `enum minoseg{jo,kozepes,rossz};`
    A *minoseg* típusú konstansok értékei: jo 0, kozepes 1, rossz 2
  - *Lebegőpontos konstansok*: 100.3, 2.567F, 1.78f (F, f - egyszeres pontosság),
    3.456L (L, l **long double** nagy pontosság)
    2e-3, 5.3e+10, 1E2, 1E-2 (E, e - exponens, a kitevő).
  - *Sztringkonstansok:* "ez egy szövegkonstans\nkét sorban"
  - *Karakterkonstansok*: 'a' egybájtos, L'2*' (két karaktert tartalmazó UniCode karakterkonstans)
- **Megjegyzések**: /* */ többsoros megjegyzés, // egysoros megjegyzés
- **A C++ nyelv alaptípusai**
  - *Adattípusok*: **bool**, **char**, **int**, **float**, **double**
  - *Típusmódosítók*: **short**, **long** – tárolási hosszat, **singed**, **unsigned** – előjel értelmezését szabályozzák.
  - *Típusminősítők*: **const** (csak olvasható), **volatile** (másik futó folyamat is megváltoztathatja).
- **Egyszerű változók definiálása**:

  <tárolási osztály>típus <típus...> változónév <= kezdőérték><,...>;

Például:

```
unsigned int i;               // előjel nélküli egész
long double x = 1.523674e12;  // nagypontosságú valós
```

Tárolási osztály: **auto, register, static, extern**;

- A függvényeken belül deklarált változók alaphelyzetben automatikus (**auto**) változók.
- A **register** változót, ha lehetséges a fordító a számítógép regiszterébe helyezi (ha nem tudja, akkor a változó a memóriában marad). A **register** változó a függvény a program futási sebességét növeli.
- A **static** változót a fordító úgy helyezi el, hogy benne van a kezdőérték, tehát a kezdőértékadás nem igényel futási időt.
- A függvényen kívül definiált változók alapértelmezés szerint globális (más modulból elérhető - **extern**) tárolási osztállyal rendelkeznek.

- **Operátorok és kifejezések**:
  - *Aritmetikai operátorok*: / , % , * , + , – (|osztás, maradék, szorzás|, |összeadás, kivonás|. Az operátorok felsorolása a műveletek precedenciája szerint történt, az azonos szintű precedenciákat a | vonal fogja össze.)
  - *Precedencia-szabály:* Műveletek precedenciája, zárójelezés.
  - *Értékadó utasítás* : jobb oldal és a bal oldal fogalma.
    a = 12.5; a = a+4; a +=4; – =, *=, /=, %=,
  - *Csoportosítási szabály* : a = b = c = 0; (jobbról balra szabály)

- **A C++ program szerkezete**

  A C++ nyelvű program egy vagy több forrásfájlban helyezkedik el, melyek kiterjesztése .CPP. A forrásprogramhoz az ún. deklarációs (include, header, fej-) állományok csatlakoznak, melyeket a `#include` előfordító utasítás segítségével építünk be a forrásállományba.
  - A C++ szabvány minden könyvtári elemet a közös *std* névterületen definiál:

    ```
    #include <iostream.h>     // helyett
    ```

    használjuk a

    ```
    #include <iostream>
    using namespace std;
    ```

  - A C++ programhoz egy ***main()*** függvény tartozik, amely kijelöli a program belépési pontját – azaz a program futása a ***main()*** függvény indításával kezdődik.
  - Az adatok beolvasására a szabványos input (***cin***), a kiírására pedig a szabványos output (***cout***) adatfolyam-objektumot használjuk. A kiírás után a soremelés kétféleképpen is biztosítható:

    ```
    cout << "\n";  vagy cout << endl;
    ```

  - Létezik még egy szabványos hibastream:

    ```
    cerr << "Hibas adat" << endl;
    ```

  - A különféle formátum kialakításához szükséges az alábbi fejállomány beszerkesztése:

    ```
    #include <iomanip>
    ```

Jelző	**Jelentése**
`ios::fixed`	fixpontos kiírás: `cout.setf(ios::fixed)`
`ios::scientific`	E hatványkitevővel, normál alakban Ha sem a *fixed*, sem a *scientific* nincs bekapcsolva, akkor az eredmény nagyságrendjétől függ, hogy fix- vagy lebegőpontos lesz a megjelenő érték,

`ios::showpoint`	A tizedespont és a tizedespont utáni vezető nullák is kiírásra kerülnek,
`ios::showpos`	A + előjel is kiíródik a pozitív egész szám előtt,
`ios::right`	jobbra tömörít,
`ios::left`	balra tömörít,
`ios::adjustfield`	A `right`, `left` váltásánál használjuk: `cout.setf(ios::left,ios::adjustfield)`
`cout.precision(n)`	*n* az értékes jegyek száma
`setw(n)`	*n* mezőszélességnek megfelelően jobbra tömörítve: `cout << setw(4) << x; cout << setw(6) << y;`
`cout << dec << x;`	decimális kiíratás,
`cout << hex << x;`	hexadecimális kiíratás,
`cout << oct << x;`	oktális kiíratás,
`cout.fill('*')`	karakterrel töltjük fel az üres helyeket jobbra, illetve balra,
`cout.width(n)`	*n* a mező szélessége.

```
//MINTA1_01
#include <iostream>
using namespace std;
int main()
{
//konstansok
 const bool log1 = true;
 const bool log2 = false;
 const int egesz = 7;
 const int oktalis = 012;
 const int hexa = 0x20;
 const double a1 = 100.3;
 const double a2= 2.56F;
 const double a3= 1.78f;
 const double a4= 2e-3;
 const double a5= 5e-6;
 const double a6= 3e+10;
 const float a7= 142.67;
 const long double
  pi = 3.14159265358979323 85;
 const char
  *s = "ez egy szovegkonstans\n"
       "ket sorban";
 const char c = 'a';
 // deklaráció
 int i;
 unsigned int j;
 short int k;
 long int t;
 double sugar = 1;
 long double ter;
 //utasítások
 cout <<"egesz: "<< egesz << endl;
 cout << oct << "oktalis: "
      << oktalis << endl;
 cout << hex << "hexadecimalis: "
      << hexa << endl;
 cout << "a1: " << a1;
 cout << "  a2: " << a2;
 cout << "  a3: " << a3 << endl;
 cout << "a4: " << a4;
 cout << "  a5: " << a5;
 cout << "  a6: " << a6 << endl;
 cout << "a7: " << a7 << endl;
 cout << "szoveg: " << s << endl;
 cout << "karakter: "<< c << endl;
 ter = sugar * sugar * pi;
 cout <<"terulet: "<< ter << endl;
 cout.precision(16);
 cout <<"terulet: "<< ter << endl;
 cin.get();
 return 0;
}
```

A program futásának eredménye:

```
egesz: 7
oktalis: 12
hexadecimalis: 20
a1: 100.3  a2: 2.56  a3: 1.78
a4: 0.002  a5: 5e-06  a6: 3e+10
a7: 142.67
szoveg: ez egy szovegkonstans
ket sorban
karakter: a
terulet: 3.14159
terulet: 3.141592653589793
```

```
//MINTA1_02
#include <iostream>
using namespace std;
int main()
{
  double a=2, b=4, c, d, e, f, g;
  c = a * b - 2;
  d = a * (b - 2);
  e = a * b / 2;
  f = (a + b)/2;
  g = a + b/2;
  cout << "c: " << c << " d: " << d
       << " e: "<< e << " f: " << f
       << " g: "<< g << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
c: 6 d: 4 e: 4 f: 3 g: 4
```

```
//MINTA1_03
#include <iostream>
using namespace std;
#include <iomanip>
int main()
{
  double a = 0.00005, b = 123.4142;
  cout << "a: " << a << " b: " << b << endl;
  //fixpontos kiiratás
  cout.setf(ios::fixed);
  cout << "a: " << a << " b: " << b << endl;
  cin.get();
  return 0;
}
```

Megjegyzés: Ha sem a *fixed*, sem a *scientific* nincs bekapcsolva, akkor a kiírás eredménye a nagyságrendtől függően fix- vagy lebegőpontos alakú lesz (lásd az első futtatási eredményt).

A program futásának eredménye:

```
a: 5e-05 b: 123.414
a: 0.000050 b: 123.414200
```

```
//MINTA1_04
#include <iostream>
using namespace std;
#include <iomanip>
int main()
{ double a = 0.00005, b = 123.4142;
  //hatványkitevős kiiratás
  cout.setf(ios::scientific);
  cout << "a: " << a
       << " b: " << b << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a: 5.000000e-05 b: 1.234142e+02
```

```
//MINTA1_05
#include <iostream>
using namespace std;
#include <iomanip>
int main()
{ double a = 1.5, b = 123.4142;
  // tizedespont utáni nullák kiírása
  cout.setf(ios::showpoint);
  // kiírás 5 értékes jegyre
  cout.precision(5);
  cout << a << endl;
  cout << b << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1.5000
123.41
```

1. Mintafeladat

A mintafeladat két egész változó beolvasását, szorzatának és hányadosának számítását kétféle módon való képzését mutatja be. (Két egész szám hányadosa egész eredményt ad!)

```
//MINTA1_06
#include <iostream>
using namespace std;
int main()
{ int a, b, szorzat;
  double hanyad1,hanyad2;
  cout << "a: "; cin >> a;
  cout << "b: "; cin >> b;
  szorzat = a * b;
  hanyad1 = a/b;
  cout << "szorzat: " << szorzat << endl;
  cout << "a/b    : " << hanyad1  << endl;
  hanyad2 = (double)a/b;
  cout << "(double)a/b: " << hanyad2;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a: 2
b: 5
szorzat: 10
a/b    : 0
(double)a/b: 0.4
```

2. Mintafeladat

Olvassunk be két valós számot (*a*,*b*)! Számítsuk ki az *a* változó *b*-edik hatványát, valamint az *a* változó gyökét!

A *cmath* könyvtár beszerkesztése szükséges a ***pow()***, és az ***sqrt()*** függvények használata miatt!

```
//MINTA1_07
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
 double a,b,hatvany, gyok;
```

Ha matematikai függvényeket használunk, akkor az alábbi fejlécállomány szükséges:

```
#include <cmath>
```

```
  cout << "a: "; cin >> a;
  cout << "b: "; cin >> b;
  hatvany = pow(a,b);
  gyok = sqrt(a);
  cout.precision(4);
  cout << "hatvanya: "
       << hatvany << endl;
  cout << "negyzetgyoke: "
       << gyok << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
a: 4
b: 3
hatvanya: 64
negyzetgyoke: 2
```

A *cin>>* sorban adatot olvasunk egy változóba. (Nem kell konverzió, sem a változó címe.)
cout<< írunk a képernyőre. Kiírható:

- "" között a szöveges információ,
- változó,
- kifejezés,
- függvényhívás,
- endl soremelés.

A ***cin.get();*** karakter leütésére vár.

Ha van olvasás a függvényben, akkor azért kell két ***cin.get();*** utasítás, mert az első az <*Enter*> karaktert nyeli le.

Feladatok

1. Olvassunk be két egész számot! Számítsuk ki a számok átlagát kétféle módon, valamint a mértani közepét! Alkalmazzuk az alábbi definíciókat (GYAK1_1):

   ```
   int a, b;
   double atlag1, atlag2, mkozep;
   ```

   Az átlag számítását kétféle módon hajtsuk végre, mivel két egész szám hányadosa egész eredményt adna:

   - A számláló típusát alakítsuk át valóssá (**double**):

     ```
     atlag1 = (double)(a+b)/2;
     ```

   - Mivel a nevező konstans, ezért oszthatunk a 2. valós számmal, így az eredmény valós lesz:

     ```
     atlag2 = (a+b)/2.;
     ```

   A program futásának eredménye:

   ```
   a: 1
   b: 2
   1. atlag: 1.5
   2. atlag: 1.5
   Mertani kozep: 1.41421
   ```

2. Olvassuk be egy téglalap két oldalának hosszát, és számítsuk ki a téglalap területét és kerületét! (GYAK1_2)

   ```
   double a,b,ter,ker;
   ```

A program futásának eredménye:

```
A teglalap a oldala: 1
A teglalap b oldala: 2

A teglalap terulete: 2
A teglalap kerulete: 6
```

3. Olvassuk be a sugár adatát! Írjunk programot, amely kiszámítja a kör kerületét, területét, a gömb felszínét és térfogatát! (GYAK1_3)

```
const double pi = 3.141592654;
double r, korker, korter, gombfelsz, gombterf;
```

$$kör kerület = 2 \cdot r \cdot \pi \qquad körterület = r^2 \cdot \pi$$

$$gömbfelszín = 4 \cdot r^2 \cdot \pi \qquad gömbtérfogat = \frac{4r^3 \cdot \pi}{3}$$

A program futásának eredménye:

```
Sugar: 1

Kor kerulete  : 6.28319
Kor terulete  : 3.14159
Gomb felszine : 12.5664
Gomb terfogata: 4.18879
```

4. Olvassuk be a kocka élének hosszát Írjunk programot, amely kiszámítja a kocka felszínét és térfogatát! (GYAK1_4)

```
double kocka_el, kocka_felsz, kocka_terf;
```

A program futásának eredménye:

```
Kocka ele: 1

Kocka felszine : 6
Kocka terfogata: 1
```

5. Olvassuk be egy derékszögű háromszög két befogóját! Számítsuk ki az átfogót, a háromszög területét és a kerületét! (GYAK1_5)

A program futásának eredménye:

```
Befogo1: 3
Befogo2: 4

Atfogo: 5
Derekszogu haromszog kerulete: 12
Derekszogu haromszog terulete: 6
```

2. Gyakorlat

Rövid elméleti összefoglaló

- **Összehasonlító és logikai operátorok**
  - Relációjelek: < , <=, >, >=, == (egyenlő), != (nem egyenlő)
  - Logikai operátorok: && (ÉS), || (VAGY), ! (NEM)

```
//MINTA2_01
#include <iostream>
using namespace std;
int main()
{
  int i = 1, j = 3, k = 12;
  if (i == 1 && j > 2)
    cout << "1. feltetel teljesult" << endl;
  else cout << "1. feltetel nem teljesult " << endl;
  if (i > 0 || k < 20)
    cout << "2. feltetel teljesult" << endl;
  else cout << "2. feltetel nem teljesult" << endl;
  if (i < -1 || j > 3 && k < 10)
    cout << "3. feltetel teljesult" << endl;
  else cout << "3. feltetel nem teljesult" << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. feltetel teljesult
2. feltetel teljesult
3. feltetel nem teljesult
```

- **Léptető operátorok**
  - ++ (increment), a változó értékének eggyel való növelése: i++; ++i;
  - -- (decrement), a változó értékének eggyel való csökkentése: j--; --j;

```
//MINTA2_02
#include <iostream>
using namespace std;
int main()
{ int i = 1, j = 5, k, n, m;
  k = n = m = 0;
  cout << "i: " << i << " j: " << j << " k: "
       << k << " n: " << n << " m: " << m << endl;
  k += 2;
  cout << "i: " << i << " j: " << j << " k: "
       << k << " n: " << n << " m: " << m << endl;
  n = i++ + --j;
  cout << "i: " << i << " j: " << j << " k: "
       << k << " n: " << n << " m: " << m << endl;
  m = ++i + j--;
  cout << "i: " << i << " j: " << j << " k: "
       << k << " n: " << n << " m: " << m << endl;
  cin.get();
  return 0;
}
```

Megjegyzés: A prefixes alak (++i, --j) használata esetén a léptetés a kifejezés kiértékelése előtt megy végbe, és a változó az új értékkel vesz részt a kifejezés kiértékelésében. A postfixes alak (i++, j--) használata esetén a léptetés a kifejezés kiértékelése után megy végbe.

A program futásának eredménye:

```
i: 1 j: 5 k: 0 n: 0 m: 0
i: 1 j: 5 k: 2 n: 0 m: 0
i: 2 j: 4 k: 2 n: 5 m: 0
i: 3 j: 3 k: 2 n: 5 m: 7
```

- **Bítműveletek** (a felsorolás a precedencia sorrendjében történt):
  - ~ 1-es komplemens (bitenkénti tagadó),
  - & bitenkénti ÉS,
  - | bitenkénti VAGY,
  - ^ bitenkénti kizáró VAGY.

```
//MINTA2_03
#include <iostream>
using namespace std;
int main()
{ unsigned int bitek1 = 0163;
  unsigned int bitek2 = 0007;
  unsigned int bitek3 = 0142;
  unsigned int bitekEs, bitekVagy, bitekKizaroVagy, negalt;
  bitekEs = bitek1 & bitek2;
  bitekVagy = bitek1 | bitek3;
  bitekKizaroVagy = bitek1 ^ bitek3;
  negalt = ~ bitek1;
  cout << oct << bitek1 << " & " << oct << bitek2
       << " = " << oct << bitekEs << endl;
  cout << oct << bitek1 << " | " << oct << bitek3
       << " = " << oct << bitekVagy << endl;
  cout << oct << bitek1 << " ^ " << oct << bitek3
       << " = " << oct << bitekKizaroVagy << endl;
  cout << " ~ " << oct << bitek1
       << " = " << oct << negalt << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
163 & 7 = 3
163 | 142 = 163
163 ^ 142 = 21
 ~ 163 = 37777777614
```

Példaként az első feladat:

```
  0001 0110 0011
& 0000 0000 0111
----------------
  0000 0000 0011 -> 3
```

- **Biteltoló műveletek**
  - << eltolás balra a << n (2^n értékkel való szorzás): `szorzas = b << 1; //kettővel való szorzás`
  - >> eltolás jobbra a >>m (2^m értékkel való osztás) `osztas = c >> 1; // kettővel való osztás`

```
//MINTA2_04
#include <iostream>
using namespace std;
int main()
{
  unsigned int bitek1 = 0004;
  unsigned int bitek2 = 0010;
  unsigned int szoroz2, oszt2;
  szoroz2 = bitek1 << 2;
  oszt2 = bitek2 >> 3;
  cout << oct << bitek1
       << " << 2 = " << oct
       << szoroz2 << endl;
  cout << oct << bitek2
       << " >> 3 = " << oct
       << oszt2 << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
4 << 2 = 20
10 >> 3 = 1
```

Első feladat:

Léptetés előtt: `0000 0100`

Léptetés után: `0001 0000`

```
  bináris 16 -> oktális 20
```

Második feladat:

Léptetés előtt: `0000 1000`

Léptetés után: `0000 0001`

```
  bináris 1 -> oktális 1
```

Megjegyzés: A kiléptetett bitek elvesznek!

- A **sizeof** operátor megadja tetszőleges változó, típus, kifejezés típusának megfelelő bájtméretet.
  - **sizeof** változó
  - **sizeof** (típusnév)

```
//MINTA2_05
#include <iostream>
using namespace std;
int main()
{
  int i = 1, ihossza, xthossza, xhossza;
  double x = 2.45;
  ihossza = sizeof i;
  xhossza = sizeof x;
  xthossza = sizeof(double);
  cout << "int merete   : " <<ihossza << " bajt" << endl;
  cout << "double merete: " <<xhossza << " bajt" << endl;
  cout << "double merete: " <<xthossza <<" bajt" << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
int merete   : 4 bajt
double merete: 8 bajt
double merete: 8 bajt
```

- **A C++ nyelv utasításai**
  - Tetszőleges kifejezés utasítás lesz, ha pontosvesszőt (;) helyezünk mögé.
  - A kapcsos zárójelek { } a logikailag összefüggő definíciókat, deklarációkat és utasításokat egyetlen összetett utasításba, blokkba fogják össze.
  - Feltételes operátor három operandussal rendelkezik: $kif_1 \; ? \; kif_2 \; : \; kif_3$

```
// a minimális adatot választja ki:
min = x < y ? x : y;
// szintén a minimalis adatot választja ki:
x < y ? min = x : min = y;
```

  - **if** (feltételes) utasítás:

if (*kifejezés*)
{
utasítások;
}
	Ha a *kifejezés* igaz (**true**, vagy a *kifejezés* értéke nem nulla), akkor a { } zárójelben lévő *utasításokat* hajtja végre.

if (*kifejezés*) *utasítás1*; **else** *utasitas2*;	Ha a *kifejezés* igaz (**true**, vagy a *kifejezés* értéke nem nulla), akkor az *utasítás1*-et hajtja végre. Ha a *kifejezés* nem igaz (**false**, vagy a *kifejezés* értéke nulla), akkor az *utasítás2*-őt hajtja végre.

```
//MINTA2_06
#include <iostream>
using namespace std;
int main()
{int a = 14, b = 5, c;
 cout << "a: " << a << "  b: " << b << endl;
 if ( a > b )
 { c = a; a = b; b = c; } // két változó tartalmát felcseréli
 cout <<"Az if utasitas vegrehajtasa utan:" << endl;
 cout << "a: " << a << "  b: " << b;
 cin.get();
 return 0;
}
```

A program futásának eredménye:

```
a: 14  b: 5
Az if utasitas vegrehajtasa utan:
a: 5  b: 14
```

```
//MINTA2_07
#include <iostream>
using namespace std;
int main()
{ int n;
  cout << "Egesz szam : "; cin >> n;
  if ( n % 2 == 0){ cout << "A szam paros!" << endl; }
  else
   { cout << "A szam paratlan!" << endl; }
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredményei:

```
Egesz szam : 12
A szam paros!

Egesz szam : 13
A szam paratlan!
```

- A **switch** utasítás (többirányú elágazatatás, a *kifejezés* csak sorszámozott típusú lehet).

```
switch (kifejezés)
{
  case konst_kifejezés:
    utasítások; break;
    ...
  default:
    utasitások;
}
```

- A **break** utasítással kiléphetünk a **switch** blokkból.

```
//MINTA2_08
#include <iostream>
using namespace std;
int main()
{
  char c;
  cout << "Kerek egy karaktert: ";
  cin >> c;
  switch (c)
  {
    case 'a': cout << "karakter: "
                 << c << endl; break;
    case 'b': cout << "karakter: "
                 << c << endl; break;
    default:
    cout << "egyeb karakter" << endl;
  }
 cin.get();  cin.get();
}
```

A *MINTA2_08* program csak az *a* vagy *b* karakter beolvasásakor írja vissza a karaktert, minden további karakter egyéb karakternek számít.

A program futásának eredményei:

```
Kerek egy karaktert: +
egyeb karakter

Kerek egy karaktert: b
karakter: b
```

Feladatok

1. Olvassunk be egy egész számot 1 és 100 között! Ha értéke páros, osszuk el kettővel, azaz toljuk el jobbra (>>) egy helyiértékkel. Ha páratlan, akkor szorozzuk meg kettővel, azaz toljuk el balra (<<) egy helyiértékkel! Az eredményeket jelenítsük meg szövegesen! (GYAK2_1)

```
int szam, szorozva, osztva;
```

A program futásának eredményei:

```
Egesz szam: 12
leptetes jobbra (osztva 2-vel): 6

Egesz szam: 13
leptetes balra (szorozva 2-vel):26
```

2. Olvassunk be egy valós számot, ha értéke pozitív, vonjuk belőle gyököt, ha negatív, akkor emeljük négyzetre! (GYAK2_2)

```
double adat, gyok, negyzet;
```

A program futásának eredményei:

```
Valos szam : 16
Negyzetgyok: 4

Valos szam : -8
Negyzete   : 64
```

3. Olvassunk be egy kor adatot, és írjuk az alábbi szövegek egyikét! (GYAK2_3)

Kor	Státusz
`kor < 6`	Nem iskolás
`≤ 6 kor ≤ 14`	Általános iskolás
`< 14 kor ≤ 18`	Gimnazista
`< 18 kor ≤ 24`	Egyetemista
`< 25 kor ≤ 62`	Dolgozó
`kor > 62`	Nyugdíjas

A program futásának eredménye:

```
Kor: 19
Egyetemista
```

4. Olvassunk be 1 és 6 közötti egy egész számot, szimulálva a kockadobást! Adjunk hibajelzést, ha rossz adatot adtunk meg! Írjuk ki a kockadobás értékét szövegesen! (GYAK2_4)

A program futásának eredményei:

```
Kockadobas (1-6) kozott: 0
Hibas adat!

Kockadobas (1-6) kozott: 4
Dobas erteke: NEGY

Kockadobas (1-6) kozott: 6
Dobas erteke: HAT
```

5. Definiáljuk az alábbi változókat, és írassuk ki a bájtban elfoglalt méretüket a **sizeof** operátor használatával! (GYAK2_5)

```
int i;  long int j; float x; double y;
```

3. Gyakorlat

Rövid elméleti összefoglaló

- **Ciklusutasítások**
  - **for** ciklus: a ciklusmagban az utasítás a megadott számsorozat mentén kerül végrehajtásra.
    for (*kezdőérték_kif; feltétel_kif; léptető_kif*)
    utasítás;

    Ha több kezdőértéket adunk a ciklus fejében, vesszővel kell azokat elválasztanunk.

```
//MINTA3_01
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int i, szam, hatvany;
  cout.setf(ios::right);
  for (i = 1, szam = 2, hatvany = 1;
      i<=6; i++)
  {
     hatvany = hatvany * szam;
     cout << setw(4) << hatvany << endl;
  }
  cin.get();
  return 0;
}
```

A *setw()* használatához szükséges az alábbi fejállomány:

`#include <iomanip>`

Írassuk ki 2 hatványait 1-6 között, jobbra tömörítve és 4 pozícióban!

A program futásának eredménye:

```
   2
   4
   8
  16
  32
  64
```

  - **while** ciklus: a ciklusmag mindaddig végrehajtódik, míg a kifejezés értéke igaz (**true**, nem nulla)
    while (*kifejezés*)
    {
     utasítás(ok);
    }

```
//MINTA3_02
  //. . .
  int i, szam = 2, hatvany;
  hatvany = 1;
  cout.setf(ios::right);
  i = 1;
  while (i <= 6)
  {
     hatvany = hatvany * 2;
     cout << setw(4)
          << hatvany << endl;
     i++;
  }
 // . . .
```

A 2 hatványainak kiírását **while** ciklussal oldjuk meg.
A **while** ciklusnál a ciklusváltozót az adott kezdőértékre állítjuk be (i = 1) a ciklus utasítása előtt. Úgy adjuk meg a feltételt (i<=6), hogy a kezdőérték mellett és a kívánt végértékig igaz legyen, mert csak akkor fogja a ciklus törzsében lévő utasításokat végrehajtani. A ciklus belsejében a ciklusváltozó értékét változtatnunk kell (i++), oly módon, hogy a ciklus elérhesse a végértékét, különben végtelen ciklust kapunk. A ciklus akkor áll le, amikor a feltétel hamis lesz.

- **do - while** ciklus : legalább egyszer végrehajtja az utasítást, majd a továbbiakban addig addig ismétli az utasítást, míg a kifejezés értéke igaz (**true**, nem nulla)
  do
  {
    utasítás;
  }**while** (*kifejezés*);

```
//MINTA3_03
//. . .
  int i, szam = 2, hatvany;
  hatvany = 2;
  cout.setf(ios::right);
  i = 1;
  do
  {
     hatvany = hatvany * 2;
     cout << setw(4)
          << hatvany << endl;
     i++;
  }while (i<=6);
  . . .
```

A 2 hatványainak kiírását **do-while** ciklussal oldjuk meg. Ez a ciklus is addig működik, míg a feltétel igaz.
Beállítjuk a ciklusváltozó kezdőértékét:
i = 1;
A ciklus törzse egyszer feltétel nélkül lefut, mert a feltételt a ciklus végén vizsgáljuk. Ugyanakkor a ciklusváltozó értékét (i++) változtatnunk kell, különben végtelen ciklust kapunk. A ciklus a feltétel hamisra válásával áll le. A feltételt úgy kell beállítanunk, hogy a kívánt kezdő- és végérték között menjen végbe a ciklus törzse.

- A **continue** utasítás
  A **continue** utasítás a ciklusok soron következő iterációját indítja a mögötte álló utasítások átlépésével.
- A **break** utasítással kiléphetünk a ciklusból.
- A **goto** utasítással a megadott címkéjű sorra adjuk át a vezérlést:
  goto *címke*:
  utasítások
  címke:
- **Definíciók bevitele az utasításokba**
  Például a **for** ciklus ciklusváltozóját definiálhatjuk:
  for (**int** *i* = *0*; *i*<*10*; *i*++)
  { }

  Megjegyezzük, hogy az így definiált változó csak a **for** utasításon belül használható. A ciklusból való kilépés után az *i* változó nem használható fel.

Feladat: Adjuk össze a ciklus változójának értékét és számláljuk meg az adatokat! A ciklusváltozó 10-es értékénél a **continue** kihagyja az iterációt és a 11-esnél folytatja. A ciklusból a 12. lépésnél lépünk ki **break** utasítással.

```
//MINTA3_04
#include <iostream>
using namespace std;
#include <iomanip>
int main()
{
  double atlag = 0;
  for (int i=1, ossz = 0, db = 0; i<15; i++)
  {  if (i == 12) break;
```

```
      if (i == 10) continue;
      ossz += i; db++;
      cout <<"i: " <<setw(2) << i <<"   osszeg: " <<setw(4) << ossz
           << "  db: " << setw(4) << db << endl;
      atlag = (double)ossz/db;
   }
   cout << endl << "atlag: " << atlag << endl;
   cin.get();
   return 0;
}
```

A program futásának eredménye:

```
i:  1   osszeg:    1   db:    1
i:  2   osszeg:    3   db:    2
i:  3   osszeg:    6   db:    3
i:  4   osszeg:   10   db:    4
i:  5   osszeg:   15   db:    5
i:  6   osszeg:   21   db:    6
i:  7   osszeg:   28   db:    7
i:  8   osszeg:   36   db:    8
i:  9   osszeg:   45   db:    9
i: 11   osszeg:   56   db:   10

atlag: 5.6
```

Feladat: Olvassunk karaktereket, és számláljuk meg, hogy hány szóközt, új sort, vízszintes *tab* karaktert ütöttünk le! A kilépés a *q* karakterrel történjen!

```
//MINTA3_05
#include <iostream>
using namespace std;
int main()
{
 char c ='Q', kod = 'q';
 int space = 0, ujsor = 0, tab = 0;
 cout << "Szoveg olvasas (q betuig) : " << endl;
 while (c != kod)
 {
   c = cin.get();
   switch (c)
   {
     case ' '  : space++; break;
     case '\n' : ujsor++; break;
     case '\t' : tab++; break;
   }
  }
  cout << "ures helyek szama: " << space << endl;
  cout << "uj sorok szama: " << ujsor << endl;
  cout << "vizszintes tab. szama: " << tab << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Szoveg olvasas (q betuig) :
Ma szep az ido.
Holnap lehet, hogy esni fog.
Nem     lesz    jo.
q
ures helyek szama: 7
uj sorok szama: 3
vizszintes tab. szama: 2
```

Feladatok

1. Olvassuk be az adatok darabszámát (max. 15) egy *db* változóba! A **for** ciklus használatával *db* darab valós számot olvasva, számláljuk meg a pozitív, negatív és a zérus adatokat! (GYAK3_1)

```
int db, i, neg=0 , poz=0, zerus=0;
double adat;
```

A program futásának eredménye:

```
Adatok szama: 4
Adat: 3
Adat: -4
Adat: 0
Adat: 6
Pozitiv adatok szama: 2
Negativ adatok szama: 1
Zerus adatok szama  : 1
```

2. Olvassunk be egy kijelentő mondatot karakterenként a pontig a **while** ciklus használatával. (A karaktereket a `c = cin.get();` utasítással olvashatjuk be.)
Számláljuk meg a magánhangzókat (a,e,i,o,u) és az egyéb karaktereket! (GYAK3_2)

```
char c ='*', kod = '.';
int a = 0, e = 0, i = 0, o = 0, u = 0, egyeb=0;
```

A program futásának eredménye:

```
Mondat beolvasasa a pontig (.) :
Ma szep az ido.
a betuk szama: 2
e betuk szama: 1
i betuk szama: 1
o betuk szama: 1
u betuk szama: 0
egyeb karakterek szama: 10
```

3. Olvassunk be egy pozitív egész számot 2 - 50 között! A helyes beolvasás ellenőrzéséhez használjunk **do** - **while** ciklust. Írjuk ki a beolvasott szám osztóit! (GYAK3_3)

```
int szam, i;
```

A program futásának eredménye:

```
Pozitiv egesz szam (max 50): 24
Osztoi:
   2
   3
   4
   6
   8
  12
```

4. Olvassunk be egy szót (max. 40 karaktert), és válasszuk szét a magánhangzókat az egyéb karakterektől (mássalhangzók)! (GYAK3_4)

```
char szo[40], maganh[40], massalh[40];
int  i=0, j=0, k=0;
```

A program futásának eredményei:

```
Szo: diofa
Maganhangzok: ioa
Massalhangzok: df

Szo: kertkapupant
Maganhangzok: eaua
Massalhangzok: krtkppnt
```

4. Gyakorlat

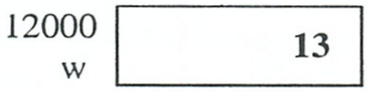
Rövid elméleti összefoglaló

- **Mutató:**

```
int w = 13, ertek;
int *poin; // a poin int-re mutató pointer (címet)tárol.
//a poin átveszi a w címét, ezért ugyanarra a memória-területre
//hivatkoznak
poin = &w;
// a *poin a poin által mutatott memória-terület tartalmát,
// azaz az adatot jelenti, így az ertek változó tartalma 13 lesz.
ertek = *poin;
```

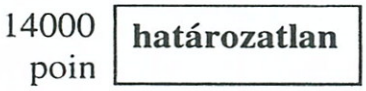
Bővebb magyarázat:

Az `int w =13;` definíció fordításakor a fordító elhelyezi a *w* változót pl. a 12000 memóriacímre, amely mögött 4 bájt szabad memória-terület van egy egész szám tárolására. Mivel a változó már a definíció során kapott kezdőértéket, a 12000 memóriacímen található tároló tartalma a 13 egész szám lesz.



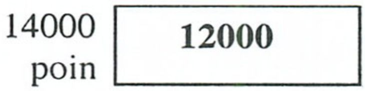
Hozzáférhetünk a *w* címéhez, az *&w* kifejezés a *w* címét jelenti, a példánkban ez 12000.

Az `int *poin;` definíció fordításakor a fordító elhelyezi a *poin* változót pl. a 14000 memóriacímre, amely mögött 4 bájt szabad memória-terület van, hogy egy memóriacím elférjen benne. Ebben az esetben a *poin* egy **int** típusra mutató címet fogad el továbbiakban, azonban kezdetben csak egy véletlenszerű értéket tartalmaz. Hibázik a program, ha nem gondoskodunk a megfelelő értékadásról.

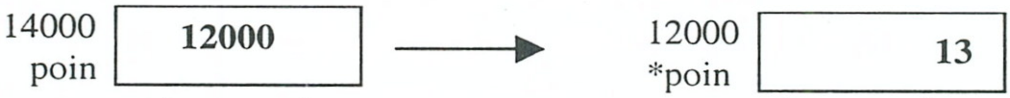


`poin = &w;` utasítás végrehajtásának hatására

a *poin* felveszi a *w* változó címét:



A 12000 cím valóban **int** adatra mutat (a példa szerint a 13 egészre).



címet tartalmaz

azt az adatot tartalmazza, ahová a *poin* mutat

Összefoglalva: a *poin* változó ebben a példában **int**-re, azaz egész számra mutató címet tárol, a **poin* pedig az adatot tartalmazza (jelen példában a 13-at), ahová a *poin* mutat.

- **Referencia** (hivatkozási) típus felhasználásával már létező változóra hivatkozva *alternatív nevet* definiálhatunk:

  típus & azonosító = változó

  ```
  float b = 5.6;
  // a referencia definiálásakor kötelező kezdőértéket adnunk.
  float &rb = b;
  ```

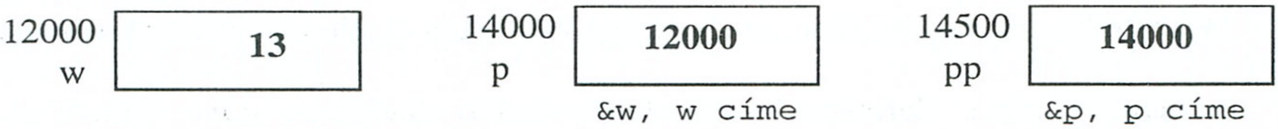
  A referencia tárolására általában nem jön létre külön változó!

- **void*** `// a C++ nyelv típus nélküli általános mutatója`

  ```
  float w;
  // float típusú változóra mutatató címet vesz át
  void *poin = &w;
  // értékadásnál felhasználói típuskonverzióval (cast) típust
  // kell a cím mellé rendelni
  *(float*)poin = 12.56;
  ```

- **Többszörös indirektségű mutatók**

  ```
  int w = 13, w1;
  int *p;    // p egy int változóra mutató pointer
  int **pp; // int* változóra mutató pointer
  p = &w; pp = &p; w1 = *p + **pp; // w1 értéke 26
  ```



A példában a **p* és a ***pp* végül a címzések által a *w* tartalmát érik el, mindkettő értéke 13 lesz, ezért a *w1* a 13+13 összegét, azaz 26-ot kap értékül.

- **Típuskonverziók:**
  A típuskonverziók egy része automatikusan, a programozó beavatkozása nélkül megy végbe, a C++ nyelv definíciójában rögzített szabályok alapján. Ezeket a konverziókat ***implicit*** vagy ***automatikus*** konverzióknak nevezzük.

```
//MINTA4_01
#include <iostream>
using namespace std;
main()
{
   int i = 12, j;
   float a = 3.43, ered1= 0;
   ered1 = i + a;
   cout << " eredmeny1: " << ered1 << endl; // eredmeny1: 15.43
   j = i + a;
   cout << " eredmeny2: " << j << endl;      // eredmeny2: 15

   cin.get();
}
```

A program futásának eredménye:

```
eredmeny1: 15.43
eredmeny2: 15
```

```
//MINTA4_02
#include <iostream>
using namespace std;
int main()
{
  int i = 12;
  // tetszőleges típusú mutató átalakítása void* mutatótípussá
  void *ip = &i;
  int *a = 0;
  a = static_cast<int*>(ip); //explicit konverzióval kell
                             // visszaalakítanunk
  cout << "*a tartalma: " << *a << endl;
 cin.get();
 return 0;
}
```

A program futásának eredménye:

```
*a tartalma: 12
```

- **Futásidejű típusazonosítás**
  A C++ nyelvben a **typeid** operátor egy **type_info** típusú objektum referenciáját adja vissza:

```
typeid (kifejezés)
typeid (típusazonosító)
```

```
//MINTA4_03
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
 int a = 2, b = 3;
 if (typeid (a) == typeid (b))
    cout << "a es a b tipusa egyezik: "
         << typeid(a).name() << endl;
 cin.get();
 return 0;
}
```

A program futásának eredménye:

```
a es a b tipusa egyezik: i
```

- **Tömbváltozó deklarálása**
  A tömb azonos típusú értékeket tud tárolni:

```
int c;
// a w tömb egész elemeket tárol, indexe 0-9-ig változhat
int w[10];
 // a z tömb valós elemeket tárol, indexe 0-49-ig változhat
double z[50];
w[0] = 12.5; // a w tömb első eleme 12.5 értéket kap.
```

- **A new operátor**
  A **new** operátor az operandusában megadott típusnak megfelelő méretű területet foglal a szabad memóriában, és a területre mutató pointert adja vissza:

`int *ip;`	*ip* **int** típusú területre mutathat, de még nem tartalmaz címet.
`ip = new int;`	A **new** operator **int** adatra mutató címet ad át az *ip* számára.
`*ip = 12;`	**ip* értékadás a mutatóval kijelölt tárolónak.
`double *tp;`	A *tp* **double** típusú pointer, amely **double** típusú változóra mutathat.
`tp = new double[20];`	A **new** operátor mögött a **double**[20] azt jelenti, hogy 20 darab **double** típusú adat számára foglaltunk helyet a memóriában, így egy 20-elemű tömböt hoztunk létre. A tömb indexelése 0-19 között lehetséges, mivel a C++ nyelvben minden tömb 0 indexű elemmel kezdődik.
`tp[0]= 1.8;`	Értékadás a tömb első, 0. indexű elemének.
`tp[19]=100.4;`	Értékadás a tömb utolsó, 19. indexű elemének.

- **A delete operátor**
  A **delete** operátor a **new** operátor által lefoglalt memória-területet szabadítja fel.

```
delete ip;    //az ip pointer által mutatott terület felszabadítása
delete[] tp;  //a tp tömb területének felszabadítása
```

A tömb helyfoglalásának vizsgálata:

```
//MINTA4_04
#include <iostream>
#include <new>
using namespace std;
int main()
{
  double *tomb;
  long int meret;
  cout << "meret: "; cin >> meret;
  try
  { tomb = new double [meret]; }
  catch(bad_alloc)
  {
    cerr <<"Nincs eleg memoria"<< endl;
    cin.get(); cin.get(); return -1;
  }
  cout<<"Sikeres a helyfoglalas"<< endl;
  // memória felszabadítása
  delete[] tomb;
  cout << "A memoria felszabaditasa\n";
  cin.get(); cin.get();
  return 0;
}
```

Kivételkezelés lehetőségei:

- A **try** blokk a próbálkozás.
- A kivételek elfogása és kezelése (**catch**).

A program futásának eredményei:

```
meret: 250
Sikeres a helyfoglalas
A memoria felszabaditasa

meret: 4000000000000
Nincs eleg memoria
```

Feladatok

1. Használjuk az alábbi definíciókat!

```
int d[40];

int db, osszeg, poz, neg, zero;
double atlag;
```

Olvassuk be az adatok számát a *db* változóba, majd töltsük fel a *d* tömböt egész adatokkal! Számítsuk ki a tömb elemeinek összegét, átlagát! Számláljuk meg a zérus, a pozitív és a negatív adatok darabszámát! (GYAK4_1)

A program futásának eredménye:

```
Adatok szama (max. 40): 6
0. adat: 2
1. adat: -4
2. adat: 8
3. adat: 0
4. adat: 12
5. adat: 0

Atlag: 3
Osszeg: 18
pozitiv elemek szama: 3
Negativ elemek szama: 1
Zerus elemek szama   : 2
```

2. Használjuk az alábbi definíciókat!

```
double a[] = {1, 2, 3, 4};
double b[] = {4, 3, 2, 1};
double Skszorzat = 0;
```

$$skalarszorzat = \sum_{i=0}^{i<M} a[i] \cdot b[i]$$

Számítsuk ki a két tömb skalárszorzatát. (GYAK4_2)

A program futásának eredménye:

```
Skalarszorzat: 20
```

3. Használjuk az alábbi definíciókat!

```
int n;
double *x, max, min, atlag;
```

Olvassuk be az adatok számát az *n* változóba! Foglaljuk le a memóriát az *x* tömb számára! Töltsük fel a tömböt adatokkal! Keressük meg a tömb legnagyobb és legkisebb elemét! A tömb elemeit osszuk el a legnagyobb és a legkisebb adatok átlagával, és jelenítsük meg a megváltozott értékeket! (GYAK4_3)

A program futásának eredménye:

```
Adatok szama: 4
0. adat: 12
1. adat: -6
2. adat: 4
3. adat: 7

Min: -6
Max: 12
Normalo tenyezo: 3
Normalt adatok
          4
         -2
    1.33333
    2.33333
```

4. Használjuk az alábbi definíciókat!

```
int db;
int x[15];
```

Olvassuk be a rendezni kívánt (max.15) adatok számát (*db*), majd olvassuk be az egész számokat az *x* tömbbe! Rendezzük a tömböt fordított sorrendbe, azaz cseréljük fel az első elemet az utolsóval, majd a másodikat az utolsó előttivel stb.! (GYAK4_4)

A program futásának eredménye:

```
Elemek szama (max. 15): 4
0. elem = 1
1. elem = 5
2. elem = 3
3. elem = 2

A forditott sorrendu tomb
2
3
5
1
```

5. Gyakorlat

Rövid elméleti összefoglaló

Felhasználó által definiált adattípusok:

- A **typedef** egy speciális tárolási osztály, mellyel érvényes típusokhoz szinonim nevet rendelhetünk.

  typedef létező_típus felhasználói_típus;

  ```
  typedef  double  tomb[10];
  tomb  x;
  ```

  A *tomb* olyan tömbtípus, amelynek 10 eleme (0-9 között indexelve) van, és minden eleme valós.

- **Struktúra – összetartozó információ tárolása**
  A struktúra több tetszőleges típusú (kivéve a **void** és függvénytípust) adatok együttese. Általános formája:

  ```
  struct nev {  // típusnév
              tipus1 tag;
              ...
          } valtozo; // változó definíció
  ```

  ```
  // diak struktúra definiálása
  struct diak {
                  char nev[20];
                  int kredit;
                  double atlag;
              };
  ```

  Definiáljunk néhány változót a *diak* típus felhasználásával:

  ```
  diak d2, *d3;
  ```

  A statikus helyfoglalású *d2* változó adattagjainak a (.) operátorral adunk értéket:

  ```
  strcpy(d1.nev,"Kiss Laszlo"); d1.kredit = 50; d1.atlag = 3.8;
  ```

  Dinamikus memória-foglalás szükséges a *d3* mutató számára, majd a mutatóval kijelölt memória-területet fel kell szabadítanunk!

  ```
  d3 = new diak;       delete d3;
  ```

  Kétféle módon adhatunk értéket a *d3* által kijelölt struktúra adattagjainak:

  - a nyíl operátor (`->`) biztosítja az adattag elérését:

    ```
    strcpy(d3->nev,"Komaromi Anna");d3->kredit = 100;
    ```

  - Amikor a pont operátort a *d3* által mutatott objektumra alkalmazzuk, a precedencia-szabályok miatt zárójelek között kell megadnunk a **d3* kifejezést:

    ```
    (*d3).atlag = 5;
    ```

  Karaktertömbnek az ***strcpy()*** függvénnyel adhatunk értéket, használatához a *cstring* fejlécállomány szükséges.

```
 //MINTA5_01
 #include <iostream>
 #include <cstring>
 using namespace std;
 struct diak{
             char nev[20];
             int kredit;
             double atlag;
            };
int main()
{ struct diak d1;    // C/C++
  diak d2, *d3;      // C++
  strcpy(d1.nev," Kiss Endre"); d1.kredit = 50;
  d1.atlag = 3.8;
  strcpy(d2.nev," "Szilard Laszlo""); d2.kredit = 80;
  d2.atlag = 4.2;
  d3 = new diak;
  strcpy(d3->nev,"Komaromi Anna");d3->kredit = 100;
  (*d3).atlag = 5;
  cout << d1.nev  << " " << d1.kredit << " "
       << d1.atlag << endl;
  cout << d2.nev  << " " << d2.kredit << " "
       << d2.atlag << endl;
  cout << d3->nev << " " << d3->kredit << " "
       << (*d3).atlag << endl;
  delete d3;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
   Kiss Endre 50 3.8
   Szilard Laszlo 80 4.2
   Komaromi Anna 100 5
```

```
//MINTA5_02
#include <iostream>
#include <cstring>
using namespace std;
struct diak{ char nev[20];
             int kredit;
             double atlag;
           }tanulo;
int main()
{ strcpy(tanulo.nev,"Kiss Zoltan");
  tanulo.kredit = 75;
  tanulo.atlag = 3.8;
  cout << tanulo.nev << " "
       << tanulo.kredit << " "
       << tanulo.atlag << endl;
  cin.get();
  return 0;
}
```

Használjuk a *diak* struktúra típussal definiált *tanulo* változót!

A program futásának eredménye:

```
Kiss Zoltan 75 3.8
```

- Használjuk a szabványos *hibastream* ***cerr*** objektumát, melyet az *iostream* fejlécállomány tartalmaz!
  `cerr << "Hibas az adat!";`

```
//MINTA5_03
#include <iostream>
using namespace std;
struct teglalap{
                 double a, b;
                 double ter, ker;
                };
int main()
{
  teglalap t;
  cout << "Teglalap oldalai: \n";
  cout << "A oldal: "; cin >> t.a;
  cout << "B oldal: "; cin >> t.b;
  if (t.a >= 0 && t.b >=0)
  { t.ter = t.a * t.b;
    t.ker = 2*(t.a + t.b);
    cout << "Tegalalap terulete: "
         << t.ter << endl;
    cout << "Tegalalap kerulete: "
         << t.ker << endl;
  }
  else
    cerr << "Hibas adat ";
  cin.get(); cin.get();
  return 0;
}
```

A struktúra deklarációját záró kapcsos zárójel mögé pontosvesszőt kell tenni.

```
struct teglalap{
                 double a, b;
                 double ter, ker;
               };
```

Olvassuk be a téglalap két oldalának hosszát! Csak akkor számítjuk ki a területet és a kerületet, ha a téglalap oldalai nagyobbak nullánál, ellenkező esetben hibajelzést adunk a ***cerr*** segítségével.

A program futásának eredménye:

```
Teglalap oldalai:
A oldal: 1
B oldal: 2
Tegalalap terulete: 2
Tegalalap kerulete: 6
```

- Struktúra feltöltése konstansokkal: `teglalap t = {1,2}; // MINTA5_04 feladat`
- A dinamikus helyfoglalású struktúra létrehozása és megszüntetése (**new** és a **delete** operátorok)

A C++ nyelvben a szükséges helyfoglalást a **new**, és a felszabadítást pedig a **delete** operátorokkal végezhetjük el.

```
//MINTA5_05
#include <iostream>
using namespace std;
struct teglalap{
                 double a, b;
                 double ter, ker;
                };
int main()
{
  teglalap *t;
  t = new teglalap;
  if (t != 0 )
  {
   cout << "Teglalap oldalai: \n";
```

A *teglalap *t* változónak a **new** operátorral foglaljunk helyet a *teglalap* típust megadva. Innen tudja a **new**, hogy a struktúra hány bájtot igényel.

```
   cout << "A oldal: "; cin >> t->a;
   cout << "B oldal: "; cin >> t->b;
   if (t->a >= 0 && t->b >=0)
   {
     t->ter = t->a * t->b;
     (*t).ker = 2*((*t).a + (*t).b);
     cout << "Tegalalap terulete: "
     << t->ter << endl;
     cout << "Tegalalap kerulete: "
     << t->ker << endl;
   }
   else { cerr << "Hibas adat "; }
  }
  else
  {
    cerr << "Nincs eleg memoria";
    cin.get(); exit(-1);
  }
  delete t;
  cin.get(); cin.get();
  return 0;
}
```

A dinamikus struktúra adattagjait kétféleképpen érhetjük el:

```
t->a
```

vagy

```
(*t).a
```

A **delete** operátorral szabadítjuk fel a lefoglalt memória-területet:

```
delete t;
```

A program futásának eredménye:

```
Teglalap oldalai:
A oldal: 3
B oldal: 4
Tegalalap terulete: 12
Tegalalap kerulete: 14
```

Feladatok

1. Használjuk fel az alábbi definíciót és deklarációt!

```
struct  adat {
                double a, b, c;
                int jel;
                double ered;
               };
int main()
{
  adat w;
  cout << "1. adat : "; cin >> w.a;
  cout << "2. adat : "; cin >> w.b;
  cout << "3. adat : "; cin >> w.c;
   . . .
```

Olvassuk be a *w* struktúra három valós adatát valamint a *jel* (0 vagy 1) értékét! Ha a *jel* értéke 0, akkor a három adat számtani közepét, ha a *jel* értéke 1, a három adat mértani közepét számítsuk ki! Ellenőrizzük az adatokat! Adjunk hibajelzést, ha nem megfelelő adatot olvastunk. (GYAK5_1)

A program futásának eredményei:

```
1. adat : 1
2. adat : 2
3. adat : 3
Valasztas
szamtani kozep(0), mertani kozep(1): 0
Szamtani kozep: 2
```

```
1. adat : 1
2. adat : 2
3. adat : 3
Valasztas
szamtani kozep(0), mertani kozep(1): 1
Mertani kozep: 1.81712
```

2. Módosítsuk az 1. feladatot, használjunk dinamikus struktúrát! (GYAK5_2)

```
struct  adat {
                double a, b, c;
                int jel;
                double ered;
              };
int main()
{
  adat *w;
  w = new adat;
  cout << "1. adat : "; cin >> w->a;
  cout << "2. adat : "; cin >> w->b;
  cout << "3. adat : "; cin >> w->c;
  . . .
}
```

3. Használjuk az alábbi definíciót és deklarációt!

```
  struct meres{
                int db;
                double MertErtek[50];
                double atlag;
                double szazalek;
                int ElteresekSzama;
              };
  meres madat;
```

Kérdezzünk rá a mérési adatok számára (max. 50), olvassuk be a mérési értékeket, és egy a *szazalek* értéket. Számítsuk ki a mérési adatok átlagát, és számláljuk meg azokat az adatokat, melyek a megadott *szazalek*nál nem térnek el jobban az átlagtól:

```
MertErtek[i] >= atlag - atlag*szazalek/100  &&
MertErtek[i] <= atlag + atlag*szazalek/100
```

(GYAK5_3)

A program futásának eredménye:

```
Meresi adatok szama: 5
0. meresi adat: 2
1. meresi adat: 3.4
2. meresi adat: 4
3. meresi adat: 5.2
4. meresi adat: 7.5

Szazalek: 20
```

```
Atlag: 4.42

3.536 es 5.304 kozotti ertekek:
2.elem: 4
3.elem: 5.2

Atlag +- szazalek elteresek szama: 2
```

4. Használjuk az alábbi definíciót és deklarációt!

```
struct szovizs{
                char szo[30];
                int a_db;
                int egyeb;
               };
int main()
{
  int i = 0;
  szovizs v;
  v.a_db= 0; v.egyeb = 0;
```

Olvassunk be egy szót és számláljuk meg, hogy mennyi 'a' betű, illetve egyéb karakter van benne! A vizsgálathoz használjunk **if** utasítást! (GYAK5_4)

A program futásának eredménye:

```
A vizsgalando szo: almafa
Az 'a' karakterek szama: 3
Egyeb karakterek szama : 3
```

5. Használjuk az alábbi definíciót és deklarációt!

```
struct szovizs{
                char szo[30];
                int a_db;
                int egyeb;
               };
int main()
{
  int i = 0;
  szovizs *pv;
  pv = new szovizs;
  pv->a_db= 0; pv->egyeb = 0;
  . . .
}
```

Módosítsuk a 4. feladatot, használjunk dinamikus adatstruktúrát! A vizsgálathoz használjuk a **swicth** utasítást! (GYAK5_5)

A program futásának eredménye:

```
A vizsgalando szo: valamifele
Az 'a' karakterek szama: 2
Egyeb karakterek szama : 8
```

6. Gyakorlat

Rövid elméleti összefoglaló

- **Függvények deklarációja és definíciója**

A saját függvényeinket mindig definiálni kell. A definíció, amelyet csak egyszer lehet megadni, a C++ programon belül bárhol elhelyezkedhet. Ha a függvény definíciója megelőzi a függvény hívásának helyét, akkor ez helyettesíti a függvény prototípusának külön megadását. A prototípus: `<visszatérési típus> Fuggvenynev(parameterek tipusa);`

```
<visszatérési típus> Fuggvénynév(paraméterek)
{ //függvény törzse
  <lokális definíciók és deklarációk >
  < utasítások>
}
```

Ha a visszatérési típus **void**, akkor a függvénynek nem kell tartalmaznia **return** utasítást, mivel nem ad vissza semmit.

Ellenkező esetben a **return** melletti kifejezés típusának meg kell egyeznie a függvény visszatérési típusával. Ebben az esetben a függvénynek ez a visszatérési értéke, vagyis a függvény működésének eredménye.

A függvény paraméterlistáján a paraméterek megadása: a paraméter típusa a paraméter nevével (vesszővel tagolva, ha több paraméter is van), például: `int n, double c` stb. A paraméterlistán adjuk meg a bemenő, és a kimenő paramétereket egyaránt. A bemenő paramétereknek kell, hogy értékük legyen, a kimenő paraméterek a függvény törzsében kapnak értéket. Csak pointerrel, vagy referenciával megadott paraméter segítségével adhatunk ki értéket.

```
int f1(int a, int b)
{
  int c;
  c = a+b; // összead
  return c;
}
```

Az *f1()* függvénynek két egész típusú bemenő paramétere van: az *a* és a *b*. A függvény feladata, hogy a két bemenő paraméter összegét adja vissza visszatérési értékként.

Lokálisan definiálunk egy egész típusú *c* változót. A *c* változóba adjuk össze az *a* és a *b* értékeket, majd a **return** utasítással adjuk vissza a *c* tartalmát.

```
int f1(int a, int b)
{
  return a+b;
}
```

A feladatot így is megoldhatjuk:
Az egyszerűbb számítást tartalmazó kifejezést a **return** mellett közvetlenül is megadhatjuk.

- **Inline függvények**

Az **inline** („soron belüli") függvényekkel helyettesíthetjük a ***#define*** makrókat. Az **inline** függvény esetén a függvény törzsét képező kód helyettesítődik be a függvényhívás helyére („kódmakró"). Általában kisméretű, gyakran hívott függvények esetén ajánlott alkalmazni ezt a megoldást. Az **inline** függvények használatának előnye, hogy a függvényhíváskor az argumentumok feldolgozása teljes körű típusellenőrzés mellett megy végbe.

```
//MINTA6_01
#include <iostream>
using namespace std;
```

```
// inline függvény definiálása
inline int min(int a, int b)
{  return a < b ? a : b; }
int main()
{
  int x = 13, y = 5, w;
  w = min(x,y);
  cout << "A kisebb adat : "
       << w << endl;
  cin.get();
  return 0;
}
```

Írjunk **inline** függvényt, amely két egész szám közül kiválasztja a kisebbet!

A *min()* **inline** függvény aktiválása.

A program futásának eredménye:

```
A kisebb adat : 5
```

- **Mutató típusú paraméter**

Kimenő paramétert a megfelelő típusra mutató pointerrel adhatjuk meg. Ez azonban nem jelenti azt, hogy bemenő paraméter nem lehet pointeres változó. A C++ nyelvben biztonságosabb a referencia típus használata.

- **Referencia típusú paraméter**

A referencia típus igazi lehetőségét a hivatkozás szerinti paraméterátadás jelenti. (A háttérben a fordító valójában címeket másol, azonban a forrásprogramból ez nem látható.) A kimenő paramétereket referenciaként adjuk meg.

Mindkét változatra nézzünk meg egy-egy egyszerű függvényt!

```
void f2(int a,int b, int& c)
{
  c = a*b; // szorzat
}
```

```
void f3(int a, int b, int* c)
{
  *c = a - b; //különbség
}
```

Az *f2()* függvény a szorzat eredményét referenciaként megadott *c* paraméterben adja vissza, míg az *f3()* függvény a különbséget az **int**-re mutató *c* pointerrel kijelölt tárolóban szolgáltatja vissza. A függvények definiálását és hívását a *MINTA6_02* mintafeladatban láthatjuk.

```
//MINTA6_02
#include <iostream>
using namespace std;
// bemenő paraméter: a , b
// kimenő érték a return mellett a visszatérési érték
int f1(int a, int b)  // definíció és a prototípus egyben
{
   int c;   // lokális definíció
   c = a+b; // összeadás
   return c;
}
//elhagyható a paraméter neve a prototipus megadásánál
void f2(int, int, int&);
void f3(int a, int b, int* c); // prototipus
int main()
{
  int x = 6, y = 3, osszeg, szorzat, kulonbseg;
  osszeg = f1(x, y);      // f1 függvény hívása
  f2(x, y, szorzat);      // f2 függvény hívása
  f3(x, y, &kulonbseg);  // f3 függvény hívása
  cout << x << " * " << y << " = " << szorzat<< endl;
  cout << x << " + " << y << " = " << osszeg<< endl;
  cout << x << " - " << y << " = " << kulonbseg<< endl;
  cin.get();
  return 0;
}
// bemenő paraméter: a, b
// kimenő paraméter a c (referencia)
void f2(int a, int b, int& c)  // f2 függvény definíciója
{
   c = a*b; //szorzat
}
// bemenő paraméter: a, b
// kimenő paraméter a c által mutatott tároló (*c)
void f3(int a, int b, int* c)  // f3 függvény definíciója
{
  *c = a - b; //különbség
}
```

Javasolt programozási módszer:

- A függvények prototípusát a ***main***() függvény előtt adjuk meg.
- A függvényeket a ***main***() függvény után dogozzuk ki.

*Megjegyzések a **main()** függvényhez:*

Az *x* és *y* változóknak már a definiálásnál kezdőértéket adunk.
Az *f1()* függvény bemenő argumentumai az *x* és az *y* változók, az eredmény az *osszeg* változóba másolódik.
Az *f2()* függvény **void** típusú, tehát nincs visszatérési értéke. Hívásakor az *x, y* a bemenő argumentumok, melyeknek van értékük, a *szorzat* változót azonban átveszi a függvény és beleírja az eredményt, azaz a két paraméter szorzatát.
Az *f3()* függvény szintén **void** típusú, tehát itt sincs visszatérési érték. A bemenő argumentumok az *x* és az *y* változók értéke. Az **int*** *c* paramétert a *kulonbseg* változó címével helyettesítettük (így kapjuk meg az eredményt).

A program futásának eredménye:

```
6 * 3 = 18
6 + 3 = 9
6 - 3 = 3
```

Számítsuk ki két valós adat számtani és mértani közepét függvénnyel, a kimenő paraméternél használjunk referenciát!

```
//MINTA6_03
#include <iostream>
using namespace std;
void SzamtaniKozep( double a, double b, double& szk);
void MertaniKozep( double a, double b, double& mk);
int main()
{
 double x = 1, y = 2, szkozep, mkozep;
 SzamtaniKozep(x,y,szkozep);
 MertaniKozep(x,y,mkozep);
 cout << "Szamtani kozep: " << szkozep << endl;
 cout << "Mertani kozep : " << mkozep << endl;
 cin.get();
 return 0;
}
void SzamtaniKozep( double a, double b, double& szk)
{
  szk = (a + b)/2;
}
void MertaniKozep( double a, double b, double& mk)
{
  mk = sqrt(a * b);
}
```

A program futásának eredménye:

```
Szamtani kozep: 1.5
Mertani kozep : 1.41421
```

Megjegyzés:

- Pointer helyett használjunk referenciát:

  int `*p` helyett **int** `&p`

  int `**pp` helyett **int** `*&pp`

- A referenciához nem készíthetünk
  - mutatót `(`**int** `&*)` és
  - referenciát `(`**int** `&&)`.

- **Tömbök átadása függvénynek**

A tömböt nem lehet érték szerint átadni a függvénynek, illetve függvényértékként megkapni. Az **int**[ ] típusú vektorargumentum **int*** típusú mutatóként adódik át a függvénynek. A függvényen belül a tömbben végrehajtott változások a függvényen kívül is megmaradnak.

```
//MINTA6_04
#include <iostream>
using namespace std;
void TombOlvas(int y[], int n);
void OsszegSzamol(int y[], int n, int& ossz);

int main()
{
  int x[10], db, osszeg = 0;
  db = 4; // a tömbnek 4 eleme lesz
  TombOlvas(x,db);
  OsszegSzamol(x,db,osszeg);
  cout << "A tomb elemeinek osszege: " << osszeg << endl;
  cin.get(); cin.get();
  return 0;
}
```

```
// A TombOlvas() függvény feladata a paraméterként kapott tömb adott
// darabszámú adattal való feltöltése.
void TombOlvas(int y[], int n)
{
  for(int i = 0; i<n; i++)
  {
    cout << i << ".elem : "; cin >> y[i];
  }
}

// Az OsszegSzamol() függvény a tömb (y) és a tömb elemeinek száma (n)
// ismeretében kiszámítja a tömb elemeinek összegét az ossz
// referencia-paraméterbe.
void OsszegSzamol(int y[], int n, int& ossz)
{
   ossz = 0;
   for(int i = 0; i<n; i++)
     ossz += y[i];
}
```

A program futásának eredménye:

```
0.elem : 4
1.elem : 1
2.elem : -2
3.elem : 0
A tomb elemeinek osszege: 3
```

- **Struktúra átadása függvénynek**

Ha a függvényben a struktúra bármelyik adattagján változtatni szeretnénk, akkor a struktúrát referenciaként kell átadnunk, például: `muvelet& v`, vagy a mutatójával.

```
//MINTA6_05
#include <iostream>
#include <cstring>
struct muvelet{
                double a,b;    // a két operandus
                double ered;   // az eredményt tárolja
                char muvjel;   // műveleti jel
               };

// A függvények prototípusai
void Feltolt(muvelet& v);
void Szamol(muvelet& v);

int main()
{
  muvelet m;
  Feltolt(m); Szamol(m);
  cout << m.a << " " << m.muvjel << " " << m.b << " = ";
  cout << m.ered;
  cin.get(); cin.get();
  return 0;
}
```

```
void Feltolt(muvelet& v)
{
  do
  {
    cout << "Muveleti jel: "; cin >> v.muvjel;
  }while (!strchr("+-*/",v.muvjel));
  cout << "1. adat: "; cin >> v.a;
  do
  {
    cout << "2. adat: "; cin >> v.b;
  }while (v.muvjel == '/' && v.b == 0);
}
void Szamol(muvelet& v)
{
  switch (v.muvjel)
  {
    case '+': v.ered = v.a + v.b; break;
    case '-': v.ered = v.a - v.b; break;
    case '*': v.ered = v.a * v.b; break;
    case '/': v.ered = v.a / v.b; break;
  }
}
```

Megjegyzések a ***main()*** *függvényhez:*

A *Feltolt()* függvényt aktiváljuk az *m* változóval, amely egy *muvelet* típusú struktúra. Ez a függvény ad értéket az *a,b*, és a *muvjel* adattagoknak. Ha a műveleti jel / (osztás), akkor a *b* adattag nem lehet 0, vizsgálat a beolvasáskor.

A *Szamol()* függvény az adatokkal feltöltött *muvelet* típusú *m* struktúra változót módosítja a kiszámított eredménnyel, majd az eredményeket kiíratjuk.

A program futásának eredménye:

```
Muveleti jel: *
1. adat: 4
2. adat: 6
4 * 6 = 24
```

- **Függvényre mutató pointer átadása függvénynek**

```
//MINTA6_06
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// fv olyan függvény, amelynek paramétere double és visszatérési
// értéke double
typedef double fv(double x);

// pfv az fv-re mutató pointer típusa
typedef fv *pfv;
```

```
// Az fg1() függvény bemenő paramétere a szög értéke fokban,
// visszatérési érték a radiánná átalakított szög szinusza.
double fg1 (double x);

// A tablazat() függvény bemenő paramétere xk - kezdőérték,
// az xv - végérték, a dx - a lépésköz
// f a függvényre mutató pointer.
void tablazat( double xk, double xv, double dx, pfv f);

int main()
{
  double kezdo=0, veg=95, lepeskoz = 10;
  tablazat(kezdo, veg, lepeskoz, fg1);
  cin.get();
  return 0;
}
double fg1 (double x)
{
  double rad, y;
    rad = x*3.141592654/180;
    y = sin(rad);
  return y;
}
void tablazat (double xk, double xv,
               double dx, pfv f)
{
  double xx, yy;
  xx = xk;
  cout.setf(ios::showpoint);
  cout.precision(4);
  cout << "  x" << "     sin(x)" << endl;
  while (xx < xv)
  {
     yy = (*f)(xx);       // azonos a f(xx) hivással
     cout << " " <<xx << "  " << yy << endl;
     xx += dx;
  }
}
```

A program futásának eredménye:

```
  x     sin(x)
 0.000  0.000
 10.00  0.1736
 20.00  0.3420
 30.00  0.5000
 40.00  0.6428
 50.00  0.7660
 60.00  0.8660
 70.00  0.9397
 80.00  0.9848
 90.00  1.000
```

Feladatok

1. Tervezzünk **inline** függvényt két pont távolságának meghatározására, írjunk hozzá ***main()*** függvényt! (GYAK6_1)

```
inline double tav(int x1,int y1,int x2,int y2)
```

A program futásának eredménye:

```
x1: 1
y1: 1
x2: 2
y2: 2
Tavolsag: 1.41421
```

2. Használjuk az alábbi definíciót és deklarációt! Írjunk függvényt a henger adatainak beolvasására, valamint a henger felszínének és térfogatának kiszámítására! Aktiváljuk a függvényeket a ***main()*** függvényben, és jelenítsük meg az eredményeket! (GYAK6_2)

```
void OlvasHengerAdatok(double& r, double& h);
void FelszTerf(double r, double h, double& felszin,
               double& terfogat);

double sugar, magassag, felsz, terf;
```

henger felszíne: $2 \cdot r^2 \cdot \pi + 2 \cdot r \cdot \pi \cdot h$

henger térfogata: $r^2 \cdot \pi \cdot h$

A program futásának eredménye:

```
Henger sugara   : 1
Henger magassaga: 1

A henger felszine : 12.5664
A henger terfogata: 3.14159
```

3. Használjuk az alábbi definíciót és deklarációt!

```
const double PI = 3.141592654;
struct henger{
               double r, h, felsz, terf;
             };

void OlvasHenger(henger& v);
void Szamol(henger& v);

henger h;
```

Írjunk *OlvasHenger()* függvényt, amely a henger adatait olvassa be; *Szamol()* függvényt, amely kiszámítja a henger felszínét és térfogatát! Mindkét függvényt aktiváljuk a ***main()*** függvényben, és az eredményeket jelenítsük meg. (GYAK6_3)

4. Használjuk fel az alábbi prototípusokat, és definíciókat!

```
void VektorOlvas(double y[], int *n);
void PozitivAtlag(double y[], int n, double& pozatlag);
void VektorKiir(double y[], int n);
int main()
{ double x[100], pozatl;
  int elemszam;
  . . .
}
```

A *VektorOlvas()* függvényben kérdezzünk rá az aktuális adatok számára *(*n)*, amely nem haladhatja meg a 100 értéket, majd töltsük fel az *y* tömböt valós adatokkal!

A *PozitivAtlag()* függvényben az *y* tömb és az *n* elemszám ismeretében számítsuk ki a tömb pozitív elemeinek átlagát, melyet a referenciaként átvett *pozatlag* paraméterben adunk vissza!

A *VektorKiir()* függvényben, jelenítsük meg az y tömb elemeit!

Írjuk meg a függvényeket és tervezzünk hozzá ***main()*** függvényt, aktiváljuk a függvényeket, majd írjuk ki a *pozatlag* eredményt! (GYAK6_4)

A program futásának eredménye:

```
A tomb elemeinek szama (1-100): 5
A tomb elemei:
0 : 12
1 : -4
2 : 0
3 : 6
4 : -2

A tomb elemei:
12
-4
0
6
-2

Pozitiv elemek atlaga: 9
```

5. Használjuk fel az alábbi prototípusokat és definíciókat!

```
typedef int vekt[10];
void TombOlvas(vekt y, int n);
void OsszegSzamol(vekt y, int n, int& ossz);

int main()
{ int db, osszeg = 0;
  vekt x;
  db = 4;  // 4 elemű lesz az x tömb
  . . .
}
```

A *vekt* felhasználói típus olyan tömböt jelent, melynek 0-9 indexelhető, 10 darab egész értékeket tároló eleme van.

A *TombOlvas()* függvényben az *n* paraméterben megadott elemszámú, *vekt* típusú *y* tömb elemeinek adjunk egész típusú értékeket!

Az *OsszegSzamol()* függvényben az *n* elemszámú *y* tömb elemeinek számítsuk ki az összegét, és azt a referenciaként átvett *ossz* paraméterben adjuk vissza!

Írjuk meg a függvényeket, tervezzünk hozzájuk ***main()*** függvényt, majd jelenítsük meg az eredményeket! (GYAK6_5)

A program futásának eredménye:

```
0.elem : -2
1.elem : 5
2.elem : 3
3.elem : 4
A tomb elemeinek osszege: 10
```

6. Használjuk fel az alábbi prototípusokat és definíciókat!

```
int* TombOlvas( int* n);
void OsszegSzamol(int* y, int n, int& ossz);

int db, osszeg = 0;
int *x;
```

A *TombOlvas()* függvényben olvassuk be a tömb elemeinek darabszámát, melyet a paraméterlistán adunk vissza. Foglaljunk helyet a dinamikus tömbnek, adjunk értéket az elemeinek, majd függvényértékként adjuk vissza a tömb kezdőcímét.

```
int* TombOlvas(int* n)
{
  int* y;
  cout << "Elemek szama: "; cin >> *n;
  y = new int[*n];
  for (int i = 0; i<*n; i++)
  {
    cout << i << ".elem : "; cin >> y[i];
  }
  return y;
}
```

Az *OsszegSzamol()* függvényben összegezzük, és adjuk vissza az összeget az *ossz* paraméterben!

Írjuk meg a függvényeket, és tervezzünk hozzájuk ***main()*** függvényt, írassuk ki az eredményeket! (GYAK6_6)

A program futásának eredménye:

```
Elemek szama: 5
0.elem : 12
1.elem : -6
2.elem : 0
3.elem : 4
4.elem : 2
A tomb elemeinek osszege: 12
```

7. Gyakorlat

Rövid elméleti összefoglaló

- **Alapértelmezés szerinti (default) argumentumok**

A C++ függvények prototípusában a paraméterekhez ún. alapértelmezés szerinti értékeket rendelhetünk. Az alapértelmezés szerinti értékekkel ellátott paraméterek *jobbról-balra* (azaz az utolsó paramétertől visszafelé) haladva folyamatosan helyezkednek el. Ha elhagyjuk a *default* paramétereket, akkor a híváskor *balról-jobbra* haladva kell megadnunk az argumentumokat, de nem hagyhatunk ki paramétert.

```
//MINTA7_01
#include <iostream>
using namespace std;
int osszeg(int x, int y = 2, int z = 3)
{ return x + y + z; }
int main()
{
  int s1, s2, s3;
  // alapertelmezés szerinti (default) argumentumok
  s1 = osszeg(1);
  cout << "osszeg(1) hivas eredmenye (2,3 default): "
       << s1 << endl;
  s2 = osszeg(5,7);
  cout << "osszeg(5,7) hivas eredmenye (3 default): "
       << s2 << endl;
  s3 = osszeg(4,3,5);
  cout << "osszeg(4,3,5) hivas eredmenye: " << s3
       << endl;
  cin.get();
  return 0;
}
```

Argumentumok:

x	y	z
1	2	3
5	7	3
4	3	5

A program futásának eredménye:

```
osszeg(1) hivas eredmenye (2,3 default): 6
osszeg(5,7) hivas eredmenye (3 default): 15
osszeg(4,3,5) hivas eredmenye: 12
```

- **Változó hosszúságú argumentumlista**

A `...` (három pont) megadásával használhatunk változó hosszúságú argumentumlistát a saját függvényeinkben, melyek feldolgozása a *cstdarg* fejlécállományban definiált makrókkal történik.

```
//MINTA7_02
#include <iostream>
#include <cstdarg>
#include <cmath>
using namespace std;
double MertaniKozep(int db,...);
int main()
{
  double mkozep;
```

```
    mkozep = MertaniKozep(2,1.0,2.0);
    cout << "Mertani kozep1 : " << mkozep << endl << endl;
    mkozep = MertaniKozep(4, 1.2, 2.5, 4.6, 8.2);
    cout << "Mertani kozep2 : " << mkozep << endl;
    cin.get();
    return 0;
}
double MertaniKozep(int db,...)
{
  va_list adat;
  double mk = 1;
  va_start(adat, db); // az első paraméter átlépése
  for (int i = 0; i<db; i++)
  {
    mk *= va_arg(adat, double); // double paraméter
  }
  mk = pow(mk,1./db);
  va_end(adat);
  return mk;
}
```

A program futásának eredménye:

```
Mertani kozep1 : 1.41421

Mertani kozep2 : 3.26154
```

Megjegyzések a változó hosszúságú argumentumlista használatához:

A függvényben *va_list* típusú változót kell deklarálni a *va_start()*, *va_arg()* és a *va_end()* makrók hívásához, mivel a makrók a *va_list* típusú mutatót használják az argumentumok eléréséhez.

Kezdőértéket ad az argumentumok eléréséhez használt mutatónak:

```
void va_start(va_list p, elso_par);
```

Az argumentumlista következő elemét adja vissza:

```
type va_arg(va_list p,type);
```

Az argumentumok felhasználása után kötelezően kell hívni:

```
void va_end(va_list p);
```

- **Rekurzív függvények alkalmazása**

A rekurzív függvények önmagukat hívják. A rekurzív megoldás általában rövidebb és áttekinthetőbb, mint az iteratív megoldás, azonban megnövekszik a feladat futási ideje és a memóriaigénye.

```
//MINTA7_03
#include <iostream>
using namespace std;
double kamatkiir(int evk, int ev,
                 double osszeg, double kamatlab);
```

```
int main()
{
  double osszeg, kamatlab, ered;
  int ev;
  cout << "Kamatszamitas " << endl;
  cout << "Betet : "; cin >> osszeg;
  cout << "Kamatlab: "; cin >> kamatlab;
  do
  {
     cout << "Ev: "; cin >> ev;
  }while (0>= ev || ev > 10);
  ered = kamatkiir(1,ev,osszeg, kamatlab);
  cin.get(); cin.get();
  return 0;
}
// A kamatkiir() rekurzív függvény adott evk (1) és ev között évenként
// kiírja a kamatos kamatot.
// A függvény további paraméterei a betét összege (osszeg) és a
// kamatláb (kamatlab).
double kamatkiir(int evk, int ev, double osszeg, double kamatlab)
{
  if (ev == 0) return osszeg;
  else
  {
    osszeg = osszeg * (1+kamatlab/100);
    cout << "Az " << evk <<". ev vegen az osszeg: " << osszeg << endl;
    return kamatkiir(evk+1, ev-1, osszeg, kamatlab);
  }
}
```

A program futásának eredménye:

```
Kamatos kamat szamitasa
Betet : 100
Kamatlab: 10
Ev: 10
Az 1. ev vegen az osszeg: 110
Az 2. ev vegen az osszeg: 121
Az 3. ev vegen az osszeg: 133.1
Az 4. ev vegen az osszeg: 146.41
Az 5. ev vegen az osszeg: 161.051
Az 6. ev vegen az osszeg: 177.156
Az 7. ev vegen az osszeg: 194.872
Az 8. ev vegen az osszeg: 214.359
Az 9. ev vegen az osszeg: 235.795
Az 10. ev vegen az osszeg: 259.374
```

- **Függvények átdefiniálása (overloading)**

A C++ nyelvben több függvényt definiálhatunk ugyanazzal a névvel, ha a definiált függvények paraméterlistája eltér egymástól. Az azonos nevű függvényváltozók közül a fordító választja ki az éppen szükségeset az argumentumok száma és típusa alapján.

Számítsuk ki két szám összegét, ha az adatok a következő típusúak lehetnek! (Az eredmény típusa is változik.)

```
  double + double → double
  int   + int → int
  double + int → double
  int + double → double
```

A feladat megvalósításához 4 függvény szükséges.

```
//MINTA7_04
#include <iostream>
using namespace std;

double Osszeg(double a, double b);
int Osszeg(int a, int b);
double Osszeg(double a, int b);
double Osszeg(int a, double b);
int main()
{
  cout << "2.5 + 4.2 = " << Osszeg(2.5,4.2) << endl;
  cout << "2 + 4 = " << Osszeg(2,4) << endl;
  cout << "2.3 + 5 = " << Osszeg(2.3,5) << endl;
  cout << "3 + 5.6 = " << Osszeg(3,5.6) << endl;
  cin.get();
  return 0;
}

double Osszeg(double a, double b)
{  return a+b; }

int Osszeg(int a, int b)
{  return a+b; }

double Osszeg(double a, int b)
{  return a+b; }

double Osszeg(int a, double b)
{  return a+b; }
```

A program futásának eredménye:

```
2.5 + 4.2 = 6.7
2 + 4 = 6
2.3 + 5 = 7.3
3 + 5.6 = 8.6
```

- **A main() függvény paraméterei és visszatérési értéke**

A *main()* függvényt általában 0,1 vagy 2 paraméterrel használjuk!

```
int main ( int argc , char* argv[ ])
```

argc - az argv tömbben található sztringek száma.
argv - karaktermutatókat (**char***) tartalmazó tömb, az argv[0] a program nevére mutat.

```
// A main() függvény paraméterként két állomány nevét várja.
//MINTA7_05
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[])
{
  if (argc != 3)
  {
    cerr << "Ket parameter szukseges: ";
    cerr << "az input és az output fajl neve" << endl;
    return EXIT_FAILURE;
  }
  cout << "A program neve: " << argv[0] << endl;
  cout << "Input fajl neve: " << argv[1] << endl;
  cout << "Output fajl neve: " << argv[2] << endl;
  cin.get();
  return EXIT_SUCCESS;
}
```

A lefordított *MAINPAR.EXE* programot a következőképpen indítjuk:

```
mainpar be.dat ki.dat
```

A program futásának eredménye:

```
A program neve: mainpar
Input fajl neve: be.dat
Output fajl neve: ki.dat
```

Megjegyzés: Az *EXIT_FAILURE* és az *EXIT_SUCCESS* szabványos konstansokat a *cstdlib* fejlécállomány tartalmazza.

- **Sztringargumentum**

A karaktersorozat-argumentumok egydimenziós karaktertömbökként (**char** s[ ]) adódnak át a függvénynek. (A **char*** mutatóval még tömörebben lehet a feladatokat megoldani.)

```
//MINTA7_06
#include <iostream>
using namespace std;
// A const  a paraméterlistán azt jelenti, hogy a paraméterek értéke
// nem változtatható meg a függvényben.
int KarSzamlal(const char s[], const char karakter);
int main()
{ int kar_db;
  char szoveg[81], kar;
  cout << "szoveg (max.80 karakter): ";
  cin.getline(szoveg,81);
  cout << "Keresett karakter: "; cin >> kar;
  kar_db = KarSzamlal(szoveg,kar);
  cout << "'" << kar << "'" << " szama a szovegben: " << kar_db;
  cin.get(); cin.get();
  return 0;
}
```

```
int KarSzamlal(const char s[], const char karakter)
{
  int db = 0;
  for (int i = 0; s[i]; i++)
  {
    if (s[i] == karakter) db++;
  }
 return db;
}
```

A program futásának eredménye:

```
szoveg (max.80 karakter): Ma szep az ido.
Keresett karakter: a
'a' szama a szovegben: 2
```

Megjegyzések:

A *getline()* hívás szóközt tartalmazó karaktersorozatot is be tud olvasni, azonban meg kell adni a karaktertömb méretét:

```
  cin.getline(szoveg,81);
```

A *KarSzamlal()* függvénynek két paramétere van: egy sztring (karaktersorozat) és egy karakter.

A függvény a *karakter* paraméterben megadott karaktert keresi az *s* sztringben, és megszámlálja, hogy hányszor fordult elő.

A ciklusban a feltétel:

```
    s[i] !='\0'
```

mivel a ciklus a sztring végét jelző 0-s bájtnál áll le.

Feladatok

1. Tervezzük meg a *Szamol()* függvényt, melynek 3 valós és két alapértékkel megadott egész paramétere van. (GYAK7_1)

   ```
   double Szamol(double x,double y,double z ,int jel=0,int h = 3);
   ```

   Ha a *jel* értéke 0, akkor a három valós szám összegét osszuk el a *h* értékével!

   Ha a *jel* értéke 1, akkor a három valós szám szorzatából vonjunk *h* értékének megfelelő gyököt!

   Ha a *jel* értéke 2, akkor a három valós szám összegét emeljük a *h* értékének megfelelő hatványra!

   Aktiváljuk a *Szamol()* függvényt, változtatva az alapértelmezett paramétereket is!

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
3. adat: 3
Eredmeny: 2
Eredmeny: 1.81712
Eredmeny: 216
```

2. Tervezzünk a *MertaniKozep()* függvényt aktiváló ***main()*** függvényt, amely változó hosszúságú argumentumlistát használ. Az első paraméter a mértani közép számításához szükséges első adat, a változó paraméterlista végét a 0-ás adat zárja. (GYAK7_2)

```
double MertaniKozep(int elso,...)
{ va_list p;
  int db = 1;
  int mk = elso, adat;
  double ered;
  va_start(p, elso);   // az első paraméter átlépése
  while (adat = va_arg(p,int))
  {
     mk *= adat; // int paraméter
     db++;
  }
  ered = pow(mk,1./db);
  va_end(p);
  return ered;
}
```

A függvényt legalább kétszer aktiváljuk különböző darabszámú adattal!

```
double mkozep;
mkozep = MertaniKozep(1,2,0);
cout << "Mertani kozep1 : " << mkozep << endl << endl;
mkozep = MertaniKozep( 1, 2, 4, 8,0);
cout << "Mertani kozep2 : " << mkozep << endl;
...
```

A program futásának eredménye:

```
Mertani kozep1 : 1.41421

Mertani kozep2 : 2.82843
```

3. Kamatos kamat számítására használjuk fel a rekurzív módon működő kamat függvényt! Írjunk hozzá ***main()*** függvényt, az eredményeket jelenítsük meg! (GYAK7_3)

```
double kamat(int ev, double osszeg, double kamatlab)
{ if (ev == 0) return osszeg;
  else
  { osszeg = osszeg * (1+kamatlab/100);
    return kamat(ev-1, osszeg, kamatlab);
  }
}
```

A program futásának eredménye:

```
Kamatos kamat szamitasa
Betet : 100
Kamatlab: 10
Ev: 10
Kamatos kammattal megnovelt osszeg: 259.374
```

4. A téglalap kerületének és területének kiszámítására írjunk függvényeket, amelyek valós és egész adatokkal is működnek! (GYAK7_4)
Használjuk fel a függvények alábbi prototípusait!

```
void TeglalapKerulet(double a, double b, double& ker);
double TeglalapTerulet(double a, double b);
void TeglalapKerulet(int a, int b, int& ker);
int TeglalapTerulet(int a, int b);
```

A program futásának eredménye:

```
Valos adatok megadasa
Teglalap a oldala: 1.5
Teglalap b oldala: 2.5
Teglalap kerulete: 8
Teglalap terulete: 3.75

Egesz adatok megadasa
Teglalap a oldala: 1
Teglalap b oldala: 2
Teglalap kerulete: 6
Teglalap terulete: 2
```

5. Tervezzünk programot, amelyben a ***main()*** függvény két paramétert vesz át, egy betű karaktert és egy szöveget! Írjunk *masol* függvényt, amely a *t* karaktertömbben kapja a szöveget, a *k* karakter tömb 0. elemében a betoldandó karaktert! A *b* tömb tartalmazza a bővített szöveget. (GYAK7_5)

```
void masol(char t[], char k[], char b[]);
```

A program futtatása:

```
Project1  - bemutat
```

A program futásának eredménye:

```
Eredeti szoveg: bemutat
Betoltando betu: -
Bovitett szoveg: b-e-m-u-t-a-t-
```

8. Gyakorlat

Rövid elméleti összefoglaló: Fájlkezelés

- **Karakter írása és olvasása**
  - `#include <fstream>` szükséges a fájlkezelő függvények használatához.
  - Állomány nyitása írásra, például:
    ```
    ofstream Fir;
    Fir.open("adatok.dat");
    ```
  - Egy karakter írása a megnyitott állományba, például:
    ```
    char kar = 'A';
    Fir.put(kar);
    ```
  - Állomány nyitása olvasásra, például:
    ```
    ifstream Folv;
    Folv.open("adatok.dat");
    ```
    Hibavizsgálat: Ha a fájlművelet alatt nem volt hiba a *fail()* tagfüggvény 0-át ad vissza, különben 1-et.
  - Egy karakter olvasása az olvasásra megnyitott állományból: `Folv.get(kar);`
  - Az állomány végének vizsgálata: az *eof* (end of file - állomány vége) tagfüggvény 0-át ad vissza, ha van még adat, 1-et, ha vége van az állománynak.
    ```
    while(!Folv.eof())
    { }
    ```
  - Ha csak vizsgálat miatt olvastunk be egy karaktert, azt visszatehetjük a kimenetre: `Folv.putback(kar);`
  - Az állományok bezárása: `Fir.close();` `Folv.close();`

```
//MINTA8_01
#include <iostream>
#include <fstream>
using namespace std;
main()
{
  char kar;
  ifstream finp;
  ofstream fout;
  finp.open("szoveg.dat");
  if (finp.fail())  // vizsgálat, létezik-e az állomány
  {
    cerr << "szoveg.dat input allomany nem letezik!\n";
    cin.get(); exit(1);
  }
  fout.open("masol.dat");
  if (fout.fail())  // vizsgálat nyitási művelet írásra
  {
    cerr << "masol.dat output allomany nyitasi hiba!\n";
    cin.get(); exit(1);
  }
 finp.get(kar);     // egy karaktert olvasunk be
 while (kar != '?') // vizsgálat a ? karakterre
 {
   fout.put(kar);   // karakterenként olvasunk
```

```
   finp.get(kar);
 }
  finp.putback(kar); // a ? karaktert visszahelyezi az input folyamra
  fout.put('!');     // ! karaktert írunk a ? karakter helyett
  finp.close(); fout.close(); // lezárjuk az állományokat
  finp.open("masol.dat"); // megnyitjuk a masol.dat állományt
  if (finp.fail())
  {
    cerr << "masol.dat input allomany nem letezik!\n";
    cin.get(); exit(1);
  }
 finp.get(kar);   //olvasunk egy karaktert az állományból
 while (!finp.eof())
 {
  cout << kar;    // kiírjuk a karaktert a képernyőre
  finp.get(kar);  // karakterenként olvasunk az állományból
 }
 finp.close();    // lezárjuk az állományt
 cin.get();
}
```

A *szoveg.dat* állomány tartalma:

Konnyu a C++ programozasi nyelv?

A feladat, hogy a ? jel helyett a *masol.dat* állományba ! jelet írjunk.

A program futásának eredménye:

```
Konnyu a C++ programozasi nyelv!
```

- **Fájlkezelés különféle típusú adatokkal**

  A különféle adatok be-, illetve kivitele az alábbi módon történik:
  - A beszúrás műveleti jellel (<<) írhatunk ki információt a kimeneti adatfolyamba.
  - A kiemelés műveleti jellel (>>) olvashatunk fájlban tárolt információt a bemeneti adatfolyamból.

  - **Karaktersorozat (szöveg) fájlkezelése**

```
//MINTA8_02
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
 char sor1[80], sor2[80], sor3[80];
 ifstream inp; // bemeneti fájlfolyam
 ofstream out; // kimeneti fájlfolyam
 out.open("szoveg.txt"); // szöveg írása a szoveg.txt állományba:
 out << "elso sor a kezdet" << endl;
 out << "masodik sor a folytatas" << endl;
 out << "harmadik sor a vege" << endl;
 out.close(); // az állomány lezárása
 inp.open("szoveg.txt"); // szöveg visszaolvasása
 inp >> sor1;
 inp >> sor2;
 inp >> sor3;
 inp.close();
 cout << sor1 << endl;
 cout << sor2 << endl;
 cout << sor3 << endl;
 cin.get();
 return 0;
}
```

Szöveg írása a *szoveg.txt* állományba:

```
elso sor a kezdet
masodik sor a folytatas
harmadik sor a vege
```

A program futásának eredménye:

```
elso
sor
a
```

Megjegyzés: A bemeneti adatfolyam a ***cin***-hez hasonlóan az elválasztó (jelen esetben a szóköz) karakterig olvasott, ezért a három sorban az első sor szóközzel elválasztott karaktersorozata jelenik meg.

- **Olvasás a sor végéig: *getline()***
  - A ***getline()*** tagfüggvény a sor végéig ('\n') olvas, de a beolvasott karaktersorozat nem tartalmazza a sort lezáró ('\n') karaktert. A függvény második paramétere a sztring hossza, harmadik paraméter a sort záró karakter ('\n').

```
ifstream& getline(char*, int, char = '\n');
```

```
//MINTA8_03
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
 char sor1[80], sor2[80], sor3[80];
 ifstream inp;
 ofstream out;
 out.open("szoveg.txt"); // szoveg.txt állomány megnyitása írásra
 out << "elso sor a kezdet" << endl;
 out << "masodik sor a folytatas" << endl;
 out << "harmadik sor a vege" << endl;
 out.close();            // állomány lezárása
 inp.open("szoveg.txt"); // szoveg.txt állomány megnyitása olvasásra
 inp.getline(sor1, sizeof(sor1)); // olvasás a sor végéig
 inp.getline(sor2, sizeof(sor2));
 inp.getline(sor3, sizeof(sor3));
 inp.close();// állomány lezárása
 cout << sor1 << endl;
 cout << sor2 << endl;
 cout << sor3 << endl;
 cin.get();
 return 0;
}
```

A program futásának eredménye:

```
elso sor a kezdet
masodik sor a folytatas
harmadik sor a vege
```

- **Numerikus adatok írása, olvasása**
  - Bemeneti állomány esetén:
  - Olvasás egy valós változóba: `inp >> a;`
  - Kimeneti állomány esetén:
  - Kiírás fixpontosan
    `out.setf(ios::fixed);`
  - A tizedespont és a tizedespont utáni vezető nullák kiírásra kerülnek:
    `out.setf(ios::showpoint);`
  - A értékes jegyek száma is beállítható
    `out.precision(3);`
  - Valós számok kiírása soremeléssel:
    ```
    out << ossz << endl;
    out << atlag << endl;
    ```

```
//MINTA8_04
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  int db = 0;
  double a, ossz = 0, atlag;
  ifstream inp;
  ofstream out;
  char InpFnev[20],OutFnev[20];
  cout <<"Input fajlnev: ";
  cin >> InpFnev; // bemeneti állomány neve (olvas)
  inp.open(InpFnev);
  if (inp.fail()) // vizsgálat: létezik-e az állomány
  {
    cout << InpFnev << " nem letezik!" << endl; cin.get(); exit(1);
  }
  cout << "Output fajlnev: ";
  cin >> OutFnev; // kimeneti állomány neve (ír)
  out.setf(ios::fixed);
  out.setf(ios::showpoint);
```

```
  out.open(OutFnev);
  if (out.fail())
  {
    cout << OutFnev << " letrehozas nem sikerult!"
    << endl;
    exit(1);
  }
  out.precision(3);
  cout << "A beolvasott adatok " << endl;
  inp >> a;  // egy valós típusú adatot olvas
  while (!inp.eof())
  {
    cout << a << " "; // kiírja az állományba
    ossz += a;        // az adatok összegét képezi
    db++;             // megszámlálja az adatok számát
    inp >> a;         // olvas egy újabb adatot
  }
  atlag = ossz/db;  // kiszámítja az adatok átlagát
  // megjeleníti a képernyőn az összeget és az átlagot
  cout << endl << "Osszeg: " << ossz << endl;
  cout << "Atlag: " << atlag << endl;
  // kiírja az állományba is
  out << ossz << endl; out << atlag << endl;
  inp.close(); // lezárja a bemeneti állományt
  out.close(); // lezárja a kimeneti állományt
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Input fajlnev: adat.txt
Output fajlnev: ered.txt
A beolvasott adatok
2.5 4.1 5.6 4.8
Osszeg: 17
Atlag: 4.25
```

Az ered.txt állomány tartalma:

```
17.000
4.250
```

- **Az ios osztály használata**

Megnyitási mód	**Hatása**
ios::app	Megnyitja az állományt (*append*) hozzáfűzési módba, és a fáljmutatót a fájl végére helyezi.
ios::ate	A fájlmutatót a fájl végére helyezi.
ios::in	A fájlt bevitelre (*input*) nyitja meg, *ifstream* esetén ez az alapértelmezés.
ios::out	A fájlt kivitelre (*output*) nyitja meg, *ofstream* esetén ez az alapértelmezés.
ios::binary	A fájlt bináris módban nyitja meg.
ios::trunc	Ha a fájl nem létezik, akkor létrehozza, ha létezik, megnyitja és felülírja.

ios::nocreate	Ha a megadott fájl nem létezik, akkor nyitási művelet hibát jelez.
ios::noreplace	Ha a fájl létezik, akkor nyitási művelet hibát jelez, kivéve ha az *ate* vagy *app* van beállítva.

```
//MINTA8_05
#include <iostream>
#include <fstream>
using namespace std;
int main()
{ ifstream inp;
  ofstream out;
  char fnev[20];
  int db;
  double adat;
  cout <<"Az allomany neve: ";
  cin >> fnev;
  out.open(fnev);
  cout << "Adatok szama: "; cin >> db;
  for (int i = 1; i<=db; i++)
  {
    cout << "Adat: "; cin >> adat;
    out << adat << endl;
  }
  out.close();
  out.open(fnev, ios::app);
  cout << "Uj adatok szama: ";
  cin >> db;
  for (int i = 1; i<=db; i++)
  {
    cout << "Adat: "; cin >> adat;
    out << adat << endl;
  }
  out.close();
  inp.open(fnev);
  cout << fnev <<" allomany tartalma:\n";
  inp >> adat;
  while (!inp.eof())
  {
     cout << adat << "  ";  inp >> adat;
  }
  inp.close();  // az állomány lezárása
  cin.get();cin.get();
  return 0;
}
```

Az állományhoz való hozzáfűzés bemutatása (***app***).

Állomány nevének beolvasása.
Állomány megnyitása kivitelre.
Adatok számának beolvasása.

Adatok fájlba írása.

Az állomány lezárása.
Állomány megnyitása ***app***, azaz hozzáfűzés módban.
Új adatok darabszámának beolvasása,

majd az új adatok beolvasása,
és fájlba írása hozzáfűzéssel.

Az állomány lezárása.

A kimeneti állomány visszaolvasása és az adatok megjelenítése.

A program futásának eredménye:

```
Az allomany neve: adat.txt
Adatok szama: 3
Adat: 2
Adat: 5
Adat: 4
Uj adatok szama: 2
Adat: 9
Adat: 12
adat.txt allomany tartalma:
2  5  4  9  12
```

- **Bináris fájlkezelés**

  Az összetett adattípusokat (tömböt és struktúrákat) kényelmesebb binárisan fájlba írni (*write*), illetve onnan olvasni (*read*).

  - **Írás**: *ostream::write()* tagfüggvénye

    ```
    ostream& write(const signed char*, int n);
    ostream& write(const unsigned char*, int n);
    ```

  - **Olvasás**: *istream::read()* tagfüggvénye

    ```
    istream& read(signed char*, int);
    istream& read(unsigned char*, int);
    ```

- **Struktúra adatainak kiírása bináris állományba**

```
//MINTA8_06
#include <iostream>
#include <fstream>
using namespace std;
struct aru{
             char azonosito[15];
             double sulya;
             int ara;
           };
int main()
{
  aru r;
  int db;
  ofstream out_aru;
  out_aru.open("aruk.dat");
  cout << "Arufajtak szama: "; cin >> db;
  for (int i=0; i<db; i++)
  {
    cout << endl;
    cout << "Aru azonositoja: ";
    cin >> r.azonosito;
    cout << "Aru sulya: "; cin >> r.sulya;
    cout << "Aru ara: "; cin >> r.ara;
    out_aru.write((char*)&r, sizeof(aru));
  }
  out_aru.close();
  cin.get(); cin.get();
  return 0;
}
```

Struktúra kiírása bináris állományba.

Rákérdezünk az árufajták számára (*db*).
Beolvassuk az áruk azonosítóját, súlyát és árát.

Kiírjuk a struktúra adatait a bináris *aruk.dat* állományba.

Lezárjuk az állományt.

A program futásának eredménye:

```
Arufajtak szama: 3

Aru azonositoja: alma
Aru sulya: 2
Aru ara: 100

Aru azonositoja: szilva
Aru sulya: 3
Aru ara: 150

Aru azonositoja: banan
Aru sulya: 4
Aru ara: 200
```

- **Struktúra adatainak visszaolvasása bináris állományból**

```
//MINTA8_07
#include <iostream>
#include <fstream>
using namespace std;
```

A program beolvassa a *MINTA8_06* által írt *aruk.dat* állományt tartalmát.

```
typedef struct aru{
                    char azonosito[15];
                    double sulya;
                    int ara;
                  };
int main()
{
  aru r;
  int db;
  double RaktarErteke = 0, OsszSuly = 0;
  ifstream inp_aru;
  inp_aru.open("aruk.dat");
  inp_aru.read((char*)&r, sizeof(aru));
  while (!inp_aru.eof())
  {
    cout << end << "Aru azonositoja: "
         << r.azonosito << endl;
    cout << "Aru sulya: " <<  r.sulya
    << endl;
    cout << "Aru ara: " << r.ara << endl;
    OsszSuly += r.sulya;
    RaktarErteke += r.sulya*r.ara;
    inp_aru.read((char*)&r, sizeof(aru));
  }
  inp_aru.close();
  cout << endl;
  cout << "Aruk ossz sulya: "
       << OsszSuly << endl;
  cout << "Aruk ossz erteke: "
       << RaktarErteke << endl; cin.get();
  return 0;
}
```

Az *aruk.dat* állományban binárisan tárolt *aru* struktúra adatainak visszaolvasása. Az állományból való olvasáskor összeadjuk az áruk súlyát, és kiszámítjuk az összértéküket.

A program futásának eredménye:

```
Aru azonositoja: alma
Aru sulya: 2
Aru ara: 100

Aru azonositoja: szilva
Aru sulya: 3
Aru ara: 150

Aru azonositoja: banan
Aru sulya: 4
Aru ara: 200

Aruk ossz sulya: 9
Aruk ossz erteke: 1450
```

Feladatok

1. Szöveges állományból olvassunk be egy karaktersorozatot a (.) pontig! Számláljuk meg magánhangzók és az üres helyek számát. (GYAK8_1)
   Használjuk az alábbi definíciókat!

```
char kar;
int osszkar=0, maganh=0, space=0;
```

2. Generáljunk 10 véletlen számot 1 és 10 között, és írjuk ki azokat a *veletlen.txt* állományba! A *veletlen.txt* állományból olvassuk vissza, és jelenítsük meg a beolvasott adatokat! Számláljuk meg a páros, a páratlan és a 4-gyel osztható számokat. A párosakat adjuk össze, számítsuk ki a páratlanoknak összegét és átlagát. (GYAK8_2)

```
szam = rand()%10+1;
```

véletlenszám-generátor kezdőértéke: `srand(unsigned(time(NULL)));`

```
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <ctime>
```

```
using namespace std;
int main()
{
  int i, db = 10, vszam, parosdb= 0, paratlandb = 0, negyoszt = 0;
  int parososszz= 0, ptosszeg = 0;
  double ptatlag;
  ifstream inp;
  ofstream out;
  . . .
  srand((unsigned)time(NULL));
  for(i = 0; i<db; i++)
  {
    vszam = rand()%10+1;
  . . .
}
```

3. Használjuk az alábbi definíciót!

```
 struct utazas{
                char uticel[15];
                int letszam;
                double ara;
                int napokszama;
              };
```

Az adatok feltöltésekor legyen az úti cél Berlin is! Írjuk ki az *utazas* struktúra adatait az *utazas.dat* bináris állományba! (GYAK8_3)

4. Olvassuk be az *utazas.dat* állomány tartalmát, és számláljuk meg az utak számát, az összes utasok számát, valamint azt, hogy hányan utaznak Berlin városába! Melyik a legdrágább és melyik a legolcsóbb út! (GYAK8_4)

A struktúra azonos 3. feladatnál megadottal, használjuk az alábbi definíciókat!

```
utazas ut;
int i, utakszama =0, osszutas = 0, berlindb = 0;
double dragajegy = 0, olcsout;
ifstream inp;
```

Az *utazas.dat* állományból táblázatosan írjuk vissza az adatokat:

```
cout << "      Uticel       Ara    Letszam" << endl;
while (!inp.eof())
{
   cout << setw(15) << ut.uticel;
   cout << setw(8) << ut.ara;
   cout << setw(7) << ut.letszam << endl;
   . . .
}
```

9. Gyakorlat

Rövid elméleti összefoglaló

- **Az Objektum-Orientált Programozás (OOP) alapjai:**
  - Az **egységbezárás** (*encapsulation*) azt jelenti, hogy az adatstruktúrákat és az adott struktúrájú adatokat kezelő függvényeket kombináljuk; azokat egy egységként kezeljük, és szabályozzuk az elérhetőségüket a külvilág számára.
  - Az **öröklés** (*inheritance*) azt jelenti, hogy a meglévő osztályokból származtatott újabb osztályok öröklik a definiálásukhoz használt alaposztályok már létező adatstruktúráit és függvényeit, ugyanakkor újabb tagokat is definiálhatnak, vagy régi tagfüggvényeket újraértelmezhetnek.
  - A **sokalakúság** (*polymorphism*) alatt azt értjük, hogy egy adott tevékenység (tagfüggvény) azonosítója ugyanaz lehet egy adott osztályhierarchián belül, ugyanakkor a hierarchia minden egyes osztályában a függvény mást valósít meg. Az ún. virtuális függvények lehetővé teszik, hogy egy adott tagfüggvény konkrét végrehajtási módja csak a program futása során derüljön ki.
  - A C++ nyelv szabványos operátorainak átdefiniálása, az ún. *operátor overloading*.
- **Statikus helyfoglalású objektumpéldány** (*MINTA9_01*)

```
class osztálynév
{
 <hozzáférés:>
   <adattagok>
 <hozzáférés:>
   <tagfüggvények>
}; // A definíciót pontosvesszővel kell zárni.
```

hozzáférés: alapértelmezés a **private**, ha nincs megadva

public – nyilvános, mindenki hozzáfér,

protected – külvilág felé privát, a származtatott osztály azonban hozzáfér,

private – a külvilág és a származtatott osztály felé is privát.

```
//MUV.H                   az osztály deklarációja
class Muvelet
{
  public:
     double a,b,ered; // a,b az adat, ered az eredmény
  public:
     void Init(double a1, double b1);
     void Szamol();
     void Kiir();
};
```

```
//MUV.CPP                 az osztály definícióját tartalmazza
#include <iostream>
using namespace std;
#include "muv.h"
```

```
void Muvelet::Init(double a1, double b1)
{ a = a1; b = b1; }
void Muvelet::Szamol() // szamtani kozep
{ ered = (a+b)/2; }
void Muvelet::Kiir()
{ cout << "eredmeny: " << ered << endl; }
```

Magyarázat:

A tagfüggvények leírását a *MUV.CPP* kiterjesztésű állományban adjuk meg. Természetesen be kell építenünk az osztály-deklarációt tartalmazó *MUV.H* állományt is.

A tagfüggvények neve előtt az osztályazonosítónak (kihez tartozik) és a hatókör operátornak (::) kell szerepelnie. Ezeket pedig a függvény visszatérési értékének típusa előzi meg.

Az *Init()* tagfüggvényben a paraméterlistán lévő paraméterekből kapnak kezdőértéket az objektumpéldányok megfelelő adattagjai.

A *Szamol()* tagfüggvény az *ered* adattagba kiszámítja az *a* és a *b* számtani közepét.

A *Kiir()* tagfüggvény kiírja az *ered* tartalmát.

```
//MAIN.CPP
#include <iostream>
#include <cstdlib>
using namespace std;
#include "muv.h"
int main()
{
  Muvelet w;
  double x,y;
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
  w.Init(x,y);  w.Szamol(); w.Kiir();
  cout << "eredmeny (w.ered): "
       << w.ered << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
eredmeny: 1.5
eredmeny (w.ered): 1.5
```

Magyarázat:

A főfüggvényt tartalmazó *MAIN.CPP* állomány is beépíti az osztályt deklaráló *MUV.H* fejlécfájlt.

A *Muvelet* osztállyal definiálunk egy *statikus* objektumpéldányt *w* néven, ezzel megtörtént számára a helyfoglalás a memóriában. Statikus objektumpéldány esetén a tagfüggvényeket a pont operátorral aktiválhatjuk.

A billentyűzetről beolvassuk az adatokat a definiált változókba, és ezekkel a változókkal aktiváljuk a *w* objektumpéldány *Init()* tagfüggvényét.

Mivel az objektumpéldány adattagjai **public** hozzáféréssel rendelkeznek, ezért a főfüggvényben hozzáférhetünk a *w.ered* tartalmához, azaz közvetlenül kiírathatjuk az értékét.

- **Információ elrejtése**. Célja az, hogy szabályozzuk az adattagokhoz, illetve a tagfüggvényekhez való hozzáférést. Általános szabály az, hogy a tagfüggvények nyilvánosak (**public**) és az adattagok pedig **private** vagy **protected** elérésűek. A **private** tagokhoz csak az osztály tagfüggvényei, míg a **protected** tagokhoz a származtatott osztály tagfüggvényei is hozzáférhetnek. (*MINTA9_02*)

```
//MUV.H
class Muvelet
{
  private:
     double a,b,ered;
  public:
     void Init(double a1, double b1);
     void Szamol();
     // GetEred() inline függvény
     double GetEred(){ return ered;}
};
```

Az adattagokhoz való hozzáférést a **private** kulcsszóval letiltjuk.
A adattagoknak a külvilág felé való elérését függvényekkel oldjuk meg. Például, az *ered* adattag értékét a *GetEred()* függvény adja vissza, melyet implicit *inline* módon adunk meg. Az *inline* megadási módot akkor használjuk, ha nem túl bonyolult a függvény törzse.

```
//MAIN.CPP
#include <iostream>
#include <cstdlib>
using namespace std;
#include "muv.h"
int main()
{ Muvelet w;
  double x,y;
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
  w.Init(x,y);
  w.Szamol();
  cout << "eredmeny (w.GetEred): "
       << w.GetEred()<< endl;
  cin.get(); cin.get();
  return 0;
}
```

A *MUV.CPP* nem változott, melyet a *MINTA9_01* feladatnál megnézhetünk.

A *Kiir()* tagfüggvényt itt már nem használjuk.

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
eredmeny (w.GetEred): 1.5
```

Megjegyzés:

Az objektumpéldányok adattagjai külön memória-területet foglalnak el, a tagfüggvények viszont csak egy példányban vannak jelen.

A *w* objektumpéldány *GetEred()* tagfüggvényével kapjuk meg az eredményt.
Csak azokat a **private** adattagok érthetők el (például az *ered*), melyekhez tagfüggvényt írunk.

- **This** pointer. Minden tagfüggvény, még a paraméter nélküliek is (**void**) rendelkeznek egy nem látható (*implicit*) paraméterrel: ez a **this**, amelyben a hívás során az aktuális objektumpéldányra mutató pointert ad át a C++, és minden adattaghivatkozás automatikusan **this**->adattag kifejezésként kerül be a kódba. (*MINTA9_03*)

```
//MUV.H
class Muvelet
{ private:
     double a,b,ered;
  public:
     void Init(double a, double b);
     void Szamol();
     double GetEred(){ return ered;}
};
```

Nem javasolt az adattagok nevével azonos nevet használni a paraméterlistán.

Példaként megnézzük, hogy mi a teendő, ha a paraméter neve azonos az adattag nevével, például az *Init()* tagfüggvény esetén.

```
//MUV.CPP
#include <iostream>
using namespace std;
#include "muv.h"
void Muvelet::Init(double a, double b)
{
  this->a = a; this->b = b;
}
void Muvelet::Szamol()
{ // szamtani kozep
  ered = (a+b)/2;
}
```

Az *Init()* tagfüggvényben úgy különböztetjük meg az adattagot az azonos nevű paramétertől, hogy az értékadásnál használjuk a **this** pointert.

A *MAIN.CPP* nem változott.

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
eredmeny (w.GetEred): 1.5
```

- **Az osztályok friend** (barát) **mechanizmusa**
  - A **friend** (barát) mechanizmus lehetővé teszi, hogy az osztály **private** és **protected** tagjait külső függvényekből is elérjük. A **friend** deklarációt az osztály deklarációján belül kell elhelyeznünk. A barát lehet egy külső függvény, de akár egy egész osztály is (annak minden tagfüggvénye). (*MINTA9_04*)

```
//OSZTALY.H
class DSzHaromszog
{
  private:
    double a,b,ter;
  public:
    void Init(double a1, double b1)
    { a = a1; b = b1; }
    void Terulet();
    double GetTerulet() { return ter; }
    friend double Atfogo( const
DSzHaromszog& h);
};
```

Feladat: egy derékszögű háromszög két befogójának (*a,b*) ismeretében számítsuk ki a területét a *Terulet()* tagfüggvénnyel. A *ter* adattag értékét a *GetTerulet()* függvénnyel adjuk vissza, így a ***main()*** függvényben is megjelenítjük azt.

Az átfogó értékét az *Atfogo()* barátfüggvénnyel számítjuk ki.

```
//OSZTALY.CPP
 #include <cmath>
 #include "osztaly.h"
  void DSzHaromszog::Terulet()
  {
    ter = a*b/2;
  }
```

terület számítása

```
  double Atfogo( const DSzHaromszog& h)
  {
    return sqrt(pow(h.a,2) + pow(h.b,2));
  }
```

Barátfüggvény: átfogó számítása. Konstans referenciaként veszi át az osztályhivatkozást.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "osztaly.h"
int main()
{
  DSzHaromszog dszh;
  dszh.Init(3.0,4.0);
  dszh.Terulet();
  cout << "Terulet: "
       << dszh.GetTerulet()
       << endl;
  // barátfüggvény aktiválása
  cout << "Atfogo: " << Atfogo(dszh)
       << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Terulet: 6
Atfogo: 5
```

- **Barátosztály alkalmazása**

A barátosztály (**class** Haromszog;) prototípusát az elé az osztály elé (**class** DSzHaromszog) kell tenni, amelynek a barátja, mivel tagadatként és **friend**-ként kell megadnunk. (*MINTA9_05*)

```
//OSZTALY.H
class Haromszog;

class DSzHaromszog
{
  friend Haromszog;
  private:
    double a,b,ter;
  public:
    void Init(double a1, double b1 )
    {  a = a1; b = b1; }
    void Terulet();
    double GetTerulet() { return ter; }
};
```

A *Haromszog* osztály barát, így minden tagfüggvényére vonatkozik a baráti viszony. Ez a kapcsolat azonban nem kölcsönös.

A *DszHaromszog* osztály tagadatai:
a , *b* a két befogó
ter terület

Tagfüggvényei:

Init() tagfüggvény kezdőértéket ad az *a*, *b* adattagoknak,

Terulet() tagfüggvény kiszámítja a derékszögű háromszög területét,

GetTerulet() függvény visszaadja a *ter* adattag értékét.

```
class Haromszog
{
 private:
   double c, ker;
 public:
  void Init(DSzHaromszog& h,double a1, double b1)
  { h.a = a1; h.b = b1;}
  void Atfogo(DSzHaromszog& h);
  void Kerulet(DSzHaromszog& h);
  double GetKer() { return ker;}
  double GetC(){ return c;}
  double GetTer(DSzHaromszog& h)
  {
    h.Terulet();
    return h.ter;
  }
};
```

A *Haromszog* osztály adattagjai:

c	az átfogó
ker	a háromszög kerülete

Tagfüggvényei:

Init() tagfüggvény a *DszHaromszog* típusú objektumot veszi át a referenciaként, így eléri az adattagokat, hogy kezdőértéket adjon nekik.

Atfogo() tagfüggvény a *DszHaromszog* típusú objektumot veszi át referenciaként, hogy átfogót tudjon számítani.

Kerulet() tagfüggvény a kerületet számítja ki.

GetKer() visszaadja a *ker* adattagot.

GetC() visszaadja a *c* adattagot.

GetTer() visszaadja a *ter* adattagot

```
//OSZTALY.CPP
 #include <cmath>
 #include "osztaly.h"
  void DSzHaromszog::Terulet() //derékszögű háromszög területe
  { ter = a*b/2; }
  void Haromszog::Atfogo(DSzHaromszog& h) // átfogó számítása
  { c= sqrt(pow(h.a,2) + pow(h.b,2)); }
  void Haromszog::Kerulet(DszHaromszog& h) // kerület számítása
  { ker = h.a+h.b+ c; }
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "osztaly.h"
int main()
{
  DSzHaromszog dszh;
  dszh.Init(3.0,4.0); // inicializálás
  dszh.Terulet(); // a terület számítása
  cout << "Terulet: "<< dszh.GetTerulet()
       << endl;
  // Haromszog osztályban deklarált
  // függvények a DSzHaromszog barátai
  Haromszog h;
  h.Init(dszh,3,4);
  cout << "Barat osztaly alkalmazasa "
       << endl;
  h.Atfogo(dszh); // átfogó számítása
  h.Kerulet(dszh);
  cout << "Atfogo: " <<  h.GetC();
  // kerület számítása
  cout <<"\nKerulet: "<< h.GetKer()<<endl;
 // eredemények kiíratása
  cout << "Terulet: " << h.GetTer(dszh)
       << endl;
  cin.get();
  return 0;
}
```

A terület kiíratása.

Haromszog osztály h statikus objektumpéldánya.
h objektumpéldány inicializálása, amely eléri a *DszHaromszog* osztály adattagjait, a *dszh* objektumpéldányon át.
A program futásának eredménye:

```
Terulet: 6
Barat osztaly alkalmazasa
Atfogo: 5
Kerulet: 12
Terulet: 6
```

Feladatok

1. Használjuk az alábbi definíciót az *OSZTALY.H* fejlécfájlban!

```
//OSZTALY.H
 class Henger
{
  private:
     double r, h, felsz, terf;
  public:
     void Init(double r1, doubfle h1)
     { r = r1; h = h1; felsz = 0; terf = 0;}
     void HengerFelszin(void);
     void HengerTerfogat(void);
     double GetFelszin(void) { return felsz; }
     double GetTerfogat(void) { return terf; }
};
```

Írjuk meg a tagfüggvényeket a henger felszínének és térfogatának kiszámítására az *OSZTALY.CPP* állományban!

Fejezzük be a *MAIN.CPP* állományban a ***main()*** főfüggvényt, olvassuk be a henger sugarát és magasságát, aktiváljuk a tagfüggvényeket, és írassuk ki az eredményeket a megfelelő tagfüggvény hívásával! (GYAK9_1)

```
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
```

```
int main()
{ Henger h;
  double sugar, magassag;
  . . .
}
```

A program futásának eredménye:

```
Henger sugara: 1
Henger magassaga: 1
Henger felszine : 12.5664
Henger terfogata: 3.14159
```

2. Használjuk az alábbi definíciót a téglatest felszínének és térfogatának számításához!

```
//OSZTALY.H
class Teglatest
{
  private:
   double a_oldal, b_oldal, c_oldal, felsz;
  public:
    void Init(double a, double b, double c)
    { a_oldal = a;
      b_oldal = b;
      c_oldal = c;
      felsz = 0;
    }
    void TeglatestFelszin(void);
    double GetFelszin(void) { return felsz; }
    friend double TeglatestTerfogat(const Teglatest& t);
};
```

Írjuk meg a *TeglatestFelszin()* tagfüggvényt, és a barátfüggvényt a térfogat számítására. A ***main()*** függvényben olvassuk be a téglatest három oldalának hosszát, aktiváljuk a tagfüggvényeket, és jelenítsük meg az eredményeket! (GYAK9_2)

```
 //MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
int main()
{
  Teglatest t;
  double a, b, c;
  . . .
}
```

A program futásának eredménye:

```
Teglatest a oldala: 1
Teglatest b oldala: 2
Teglatest c oldala: 3
Teglatest felszine: 22
Teglatest terfogata: 6
```

3. Használjuk fel az alábbi definíciót!

```
//OSZTALY.H
class Vektor
{
  private:
     int db;
     double x[10];
     double osszeg, atlag;
  public:
     void Init(int db1);
     void VektorOsszeg(void);
     void VektorAtlag(void);
     double GetOsszeg(void) { return osszeg; }
     double GetAtlag(void) { return atlag; }

     friend double VektorMaxKeres(Vektor& v);
     friend double VektorMinKeres(Vektor& v);
};
```

Dolgozzuk ki az *OSZTALY.CPP* állományban a tagfüggvényeket a vektor adatainak beolvasására, az elemek összegének, átlagának kiszámítására, valamint a két **friend** függvényt a *max*, és a *min* adat megkeresésére!

A *MAIN.CPP* állományban olvassuk be a vektor elemeinek darabszámát, ellenőrizzük, hogy az adat 1 és 10 közé esik-e. Aktiváljuk a tagfüggvényeket és jelenítsünk meg az eredményeket! (GYAK9_3)

A program futásának eredménye:

```
Elemek szama (max 10): 5
0. adat: 1
1. adat: -3
2. adat: 5
3. adat: 6
4. adat: 0

Vektor osszege  : 9
Vektor atlaga   : 1.8
Vektor max adata: 6
Vektor min adata: -3
```

10. Gyakorlat

Rövid elméleti összefoglaló

- **Konstruktor használata.**
  A C++ programban az objektumok inicializálását speciális tagfüggvények, a konstruktorok végzik.
  A konstruktor általános tulajdonságai és feladatai:
  - Olyan speciális tagfüggvény, amelynek neve megegyezik az őt tartalmazó osztály nevével.
  - Nincs visszatérési értéke (**void** típusú sem). A deklarációban nem használhatjuk a **virtual**, **static**, **const**, **mutable** és a **volatile** kulcsszavakat. Nem készíthető mutató a konstruktorhoz.
  - A neve átdefiniálható (túlterhelhető).
  - Nem öröklődik, nem lehet virtuális.
  - Nem foglal memóriát a létrejövő objektum számára.
  - Az objektumpéldány létrehozása után nem hívható.
  - Ha az objektum mutatót tartalmaz, akkor a konstruktorban kell gondoskodnunk a mutató által kijelölt terület létrehozásáról (**new**).
  - A konstruktor feladata a már lefoglalt memóriaterület inicializálása.
  - Egy osztálynak többféle konstruktora lehet: alapértelmezett, paraméteres, másoló stb.
  - Egy speciális konstruktor az ún. másoló konstruktor (*copy constructor*). Másoló konstruktorok működnek akkor, amikor a definícióban egy másik objektummal inicializálunk.
  - Ha nem definiálunk konstruktort egy adott osztályhoz, akkor a C++ fordító a saját alapértelmezése szerinti változatát használja.

```
//MUV.H
class Muvelet
{
  private:
     double a,b,ered;
  public:
     Muvelet(double a1, double b1);
     void Szamol();
     double GetEred(){ return ered;}
};
```

(MINTA10_01)

Számtani közép számítása, konstruktor használata.

A *Muvelet* konstruktornak két (*a1*,*b1*) paramétere van.

```
//MUV.CPP
#include <iostream>
using namespace std;
#include "muv.h"
Muvelet::Muvelet(double a1, double b1)
{
  a = a1; b = b1;
}
void Muvelet::Szamol()
{ // számtani közép számítása
  ered = (a+b)/2;
}
```

A *Muvelet* konstruktor az (*a1*,*b1*) paraméterekkel ad kezdőértéket az *a* és a *b* adattagoknak.

```
//MAIN.CPP
#include <iostream>
#include <cstdlib>
using namespace std;
#include "muv.h"
int main()
{
  double x,y;
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
  Muvelet w(x,y);
  w.Szamol();
  w.Kiir();
  cout << "Szamtani kozep: "
       << w.GetEred()<< endl;
  cin.get(); cin.get();
  return 0;
}
```

A *Muvelet* osztály *w* statikus objektumpéldányának definiálása és inicializálása a konstruktorral.

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
Szamtani kozep: 1.5
```

- **Dinamikus objektumpéldányok létrehozása és megszüntetése.**
  A C++ nyelvben az objektumok működéséhez szükséges, a **new** operátorral lefoglalt memória-területek felszabadításáról, (fájlok lezárása stb.) egy speciális tagfüggvényben, a destruktorban kell gondoskodnunk.
  A **destruktor** tulajdonságai és feladatai:
  - Nevét a hullám karakterrel (~) egybeírt osztálynévként kell megadni.
  - Nincs visszatérési értéke (**void** típusú sem). A deklarációban nem szerepeltethetjük a **static, const**, **mutable** és a **volatile** kulcsszavakat. Nem készíthető mutató a destruktorhoz.
  - Nincsenek paraméterei.
  - Nem definiálható át (nem terhelhető túl), azonban az öröklés során lecserélhető.
  - A **new** operátorral létrehozott objektumok esetén a destruktor aktivizálása csak a **delete** operátor használatával lehetséges.
  - A destruktor lehet virtuális (**virtual**). Ha az alaposztály destruktora virtuális, akkor a származtatott osztály destruktora is automatikusan virtuális lesz, de a konstruktorok nem lehetnek virtuálisak.
  - Explicit módon is hívható.

```
//MUV.H
class Muvelet
{
  private:
    double *a, *b; // adatok
    double ered;   // eredmény
  public:
    Muvelet(double a1, double b1);
    ~Muvelet();
    void Szamol();
    double GetEred(){ return ered;}
};
```

(*MINTA10_02*)
Számtani közép számítása, dinamikus adattagok, konstruktor és desrtuktor használata.
az adatok **double*** mutatók

konstruktor
destruktor

```
//MUV.CPP
#include <iostream>
using namespace std;
#include "muv.h"
Muvelet::Muvelet(double a1, double b1)
{
  a = new double; b = new double;
  *a = a1; *b = b1;
}
Muvelet::~Muvelet()
{
  delete a; delete b;
}
void Muvelet::Szamol()
{
  // számtani közép számítása
  ered = (*a + *b)/2;
}
```

A konstruktorban a **a* és a **b* tárolóknak a **new** operátorral foglalunk helyet a memóriában, majd inicializáljuk azokat.

A destruktorban a **delete** operátorral felszabadítjuk az *a* és a *b* mutatókkal kijelölt területet.

```
//MAIN.CPP
#include <iostream>
#include <cstdlib>
using namespace std;
#include "muv.h"

int main()
{
  Muvelet *pw;
  double x,y;
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
  pw= new Muvelet(x,y);
  pw->Szamol();
  cout << "Szamtani kozep: "
       << pw->GetEred() << endl;
  delete pw;
  cin.get(); cin.get();
  return 0;
}
```

Dinamikus objektumpéldány definiálása (**pw*).

A **new** operátorral hozzuk létre a szükséges memória területet a **pw* dinamikus objektumpéldány számára, és egyben meghívódik a ***Muvelet()*** konstruktor a megadott argumentumokkal. A dinamikus objektumpéldány a tagokat a –> operátorral érheti el.
A **delete** operátorral töröljük a **pw* dinamikus objektumpéldány által lefoglalt memória-területet.

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
Szamtani kozep: 1.5
```

- **A másoló konstruktor, az alapértelmezett és a paraméteres konstruktor** működésének bemutatása.
  Ha bármilyen saját konstruktort készítünk, akkor már a paraméter nélküli alapértelmezett (*default*) konstruktor többé nem lesz elérhető. Saját másoló konstruktort általában akkor írunk, ha valamilyen dinamikus tárterület tartozik az osztály példányaihoz.

```
//MUV.H
class Muvelet
{ private:
    double a,b,ered;
```

(*MINTA10_03*)
Számtani közép számítása, háromféle konstruktor használata.

```
  public:
    Muvelet(){ a = 1; b = 1;}
    Muvelet(double a1, double b1);
    Muvelet(const Muvelet& v);
    void Szamol();
    double Eredmeny() { return ered; }
};
```

A paraméter nélküli konstruktor
Paraméteres konstruktor
Másoló konstruktor

```
//MUV.CPP
#include <iostream>
using namespace std;
#include "muv.h"
Muvelet::Muvelet(double a1,
                 double b1)
{  a = a1; b = b1; }
```

Paraméteres konstruktor

```
Muvelet::Muvelet(const Muvelet& v)
{ a = v.a;  b = v.b; ered = 0; }
void Muvelet::Szamol()
{ // számtani közép számítása
  ered = (a+b)/2;
}
```

A másoló konstruktor átmásolja a paraméterlistán megadott objektumpéldány adattagjainak értékét.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "muv.h"
int main()
{
  Muvelet w1;
  double x,y;
 // default konstruktor hivasa
  w1.Szamol();
  cout << "Eredmeny: "
       << w1.Eredmeny() << endl;
```

A *w1* statikus objektumpéldány definiálása, itt a paraméter nélküli konstruktor hívódik meg.

```
  Muvelet *pw2 = new Muvelet(w1);
  pw2->Szamol();
  cout << "Eredmeny: "
       << pw2->Eredmeny() << endl;
  delete pw2;
```

A **pw2* dinamikus objektumpéldány definiálása és a másoló konstruktor *w1* argumentummal hívódik meg.

```
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
```

Beolvassuk az adatokat az *x* és az *y* változókba.

```
  Muvelet w2 = Muvelet(x,y);
```

A *w2* statikus objektumpéldány definiálása – a paraméteres konstruktor az *x,y* paraméterrel ad kezdőértéket ad adattagoknak.

```
  // copy konstruktor hivasa
  Muvelet *pw3 = new Muvelet(w2);
  pw3->Szamol();
  cout << "Eredmeny: "
       << pw3->Eredmeny() << endl;
```

A **pw3* dinamikus objektumpéldány definiálása – a másoló konstruktor a *w2* objektumpéldány adattagjait másolja át.

```
  delete pw3;
  cin.get(); cin.get();
  return 0;
}
```

A **delete** a megfelelő dinamikus objektumpéldányt törli a memóriából a destruktor hívásával. Ha a destruktor nincs definiálva, mivel nincsenek mutató típusú adattagok, akkor az alapértelmezett destruktor hívódik meg.

A program futásának eredménye:

```
Eredmeny: 1
Eredmeny: 1
1. adat: 2
2. adat: 3
Eredmeny: 2.5
```

Feladatok

1. Használjuk az alábbi definíciót! (GYAK10_1)

```
class Teglalap
{
  private:
    double a_oldal, b_oldal, terulet, kerulet;
  public:
    Teglalap(){a_oldal = 1;b_oldal = 1;}      // paraméter nélküli
    Teglalap(double a1, double b1);           // paraméteres
    Teglalap (const Teglalap& tg);            // másoló konstruktor
    void TeglalapKerulet(void);
     void TeglalapTerulet(void);
    double GetKerulet(void){ return kerulet; }
    double GetTerulet(void){ return terulet; }
};
```

A három objektumpéldányt (*t1*, *t2* és *t3*) különböző konstruktorral inicializáljuk!

2. Használjuk az alábbi definíciót! (GYAK10_2)

```
//OSZTALY.H
class Teglalap
{
  private:
    double *a_oldal, *b_oldal, terulet, kerulet;
  public:
    Teglalap(double a1, double b1); // parameterezett
    ~Teglalap();
    void TeglalapTerulet(void);
    void TeglalapKerulet(void);
    double GetKerulet(void) { return kerulet; }
    double GetTerulet(void) { return terulet; }
};
```

A ***main()*** függvényben az alábbi módon hozzuk létre és inicializáljuk a dinamikus objektumpéldányt.

```
Teglalap *t1 = new Teglalap(a,b);
```

Írjunk destruktort, és hívjuk meg a **delete** kulcsszóval!

```
delete t;
```

3. Használjuk az alábbi definíciót! (GYAK10_3)

```
//OSZTALY.H
class Meres
{
  private:
    int db;
    double *y;
    double maxadat;
  public:
    Meres(void);
    ~Meres();
    void NormalSzamitas(void);
    void Kiiras(void);
};
```

Dolgozzuk ki az alábbi tagfüggvényeket:

A konstruktor olvassa be a mérési adatok számát a *db* adattagba, majd foglaljon memóriát az *y* tömb számára, és olvassa be a mérési adatokat!

A destruktor szabadítsa fel a dinamikus tömb által lefoglalt memória-területet!

A *NormalSzamitas()* tagfüggvény a *maxadat* adattagba keresse ki a mérési adatok közül a legnagyobbat, majd ezzel az értékkel ossza el a mérési adatokat.

A *Kiiras()* tagfüggvény írja ki a tömb indexét, és a hozzá tartozó normált mérési adatot táblázatos formában!

A ***main()*** függvényben aktiváljuk a tagfüggvényeket és jelenítsük meg az eredményt!

A program futásának eredménye:

```
Elemek szama: 5
0. adat: 10
1. adat: 15
2. adat: 27
3. adat: 4
4. adat: 17

legnagyobb ertek: 27

normalt adatok

0  0.37037
1  0.555556
2  1
3  0.148148
4  0.62963
```

11. Gyakorlat

Rövid elméleti összefoglaló

- **Operátorok túlterhelése (operator overloading)**
  - A C++ nyelvben adott a lehetőség arra, hogy a programozó által definiált függvényt szabványos operátorhoz kapcsoljuk, ezzel kibővítve az operátor működését.
  - Az operátorfüggvények legalább egyik paraméterének **class**, vagy **struct** típusúnak kell lennie.
  - Az operátorfüggvény deklarációja:

    típus **operator** *op*(*paraméterlista*);

  Nem definiálhatók át az alábbi operátorok:
  - tagkiválasztás (.)
  - indirekt tagkiválasztás (.*)
  - hatókörfeloldás (::)
  - feltételes (?:)
  - **sizeof**
  - **typeid**

Műveletek komplex számokkal. (*MINTA11_01*)

```
//OSZTALY.H
class Complex
{
  private:
    double real; // valós rész
    double imag; // képzetes rész
  public:
    Complex(){ real = 0; imag = 1; } //paraméter nélküli konstruktor
    Complex(double r, double im) // paraméteres konstruktor
    { real = r; imag = im; }
    // a + operátor túlterhelése
    friend Complex operator + (Complex c1, Complex c2);
    // a * operátor túlterhelése
    friend Complex operator * (Complex c1, double d);
    void Kiir();
};
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
// + operátor két komplex számot ad össze
Complex operator + (Complex c1, Complex c2)
{
  return Complex(c1.real+c2.real, c1.imag + c2.imag);
}

// * operátor a komplex számot egy valós konstanssal szorozza meg
Complex operator * (Complex c1, double d)
{
  return Complex(c1.real*d, c1.imag *d);
}
```

```
// a komplex szám valós és képzetes részének kiírása
void Complex::Kiir()
{
  cout << "Valos: " << real << "  ";
  cout << "Kepzetes: " << imag << endl;
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{
  Complex cx1(1,1), cx2(2,2), cx3, cx4, cx5;
  cx3 = cx1 + cx2;
  cx3.Kiir();
  cx4 = cx2*3;
  cx4.Kiir();
  cx5 = cx1*2 + cx2;
  cx5.Kiir();
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Valos: 3  Kepzetes: 3
Valos: 6  Kepzetes: 6
Valos: 4  Kepzetes: 4
```

- Az operátorfüggvényeket általában osztályon belül definiálhatjuk. Megjegyzés:
  - Az =, (), [] és -> operátorokat csak nem statikus tagfüggvénnyel lehet átdefiniálni.
  - A **new** és a **delete** operátorok túlterhelése **csak statikus** tagfüggvénnyel történik!
  - Minden más operátor megadható tagfüggvényként, vagy az osztály **friend** függvényeként.
- Az operátorok lehetnek
  - egyoperandusúak,
  - kétoperandusúak.

A **new** és a **delete** operátorok átdefiniálása (túlterhelése). (*MINTA11_02*)

```
//OSZTALY.H
#include <new>
class Adatok
{
  public:
    int k;        // egész típusú adattag
    double w;     // valós típusú adattag
  public:
    void* operator new(size_t n);
    void operator delete(void *p, size_t n);
};
```

```
//OSZTALY.CPP
#include "Osztaly.h"
#include <new>

// A new operátor túlterhelése.
void* Adatok::operator new(size_t n)
{
  cout << "letrehozas" << endl;
  return ::new char[n]; // az eredeti new hívása
}

// A delete operátor túlterhelése.
void Adatok::operator delete(void *p, size_t n)
{
  cout << "megszuntetes" << endl;
  ::delete p;           // az eredeti delete hívása
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"

int main()
{ Adatok *p = new Adatok;
  p->k = 2; p->w = 4.5;
  cout << "Tagadatok: " << p->k << "  " << p->w << endl;
  delete p;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
letrehozas
Tagadatok: 2  4.5
megszuntetes
```

- **Felhasználó által definiált típuskonverzió**

  A C++ nyelv lehetővé teszi, hogy a saját adattípusunkhoz (osztály) típuskonverziókat rendeljünk. A típuskonverziót végrehajtó tagfüggvény:

  operator *típus ()*;

  A függvény visszatérési értékének típusa megegyezik a függvény nevében használt típussal. A típuskonverziós operátor csak visszatérési típus és paraméterlista nélküli tagfüggvény lehet.

- **Az osztályok bővítése input/output műveletekkel.**

  A C++ nyelv lehetővé teszi, hogy az osztályokon alapuló adatfolyamoknak megtanítsuk saját osztályunk objektumainak kezelését. A bevitelhez az ***istream***, a kivitelhez az ***ostream*** adatfolyamosztályt, az input/output műveletek végzéséhez, pedig az átdefiniált >> és << operátorokat használjuk. A műveletek saját átdefiniált változatát **friend** operátorfüggvényként kell elkészítenünk.

Feladat: Az *Adatok* osztályhoz rendeljünk egy **double** operátort, melyet *inline* módon dolgozunk ki, valamint definiáljuk át a << kiíró és a >> beolvasó operátorokat, hogy az osztály adattagjait kiírja ki és beolvassa! (*MINTA11_03*)

```
//OSZTALY.H
#include <iostream>
using namespace std;
class Adatok
{
  private:
    int a,b;
  public:
    Adatok () { a = 0; b = 0;  }
    Adatok ( int a1, int b1 ) { a = a1; b = b1; }
    operator double() { return (double)a/b; }
    friend istream & operator >> (istream&, Adatok&);
    friend ostream & operator << (ostream&, const Adatok&);
};
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
// A >> művelet az osztály a és b adattagjait olvassa be.
istream & operator >> (istream& is, Adatok& x)
{
  cout << "\nosztaly adattagjai\n";
  cout << "a: "; is >> x.a; cout << "b: ";is >> x.b;
}
// A << kiíró operátor is az osztály a, b adattagjait írja ki.
ostream & operator << (ostream& os, const Adatok& x)
{ os << "adattagok: " << x.a << "  " << x.b << endl; }
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{ Adatok w;
  cin >> w; cout << w << endl;
  cout << "adattagok hanyadosa: " << double(w) << endl;
  cout << "adattagok hanyadosa: " << double(Adatok(4,5));
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
osztaly adattagjai
a: 5
b: 2
adattagok: 5  2

adattagok hanyadosa: 2.5
adattagok hanyadosa: 0.8
```

- **A C++ nyelv bővítése saját típussal**
  Az adattípushoz osztályt kell készítenünk megfelelő adattagokkal, és gondoskodnunk kell a kezdőérték adási lehetőségekről. A szükséges operátorok túlterhelésével a típusunk ugyanúgy használható, mint a nyelv alaptípusai.

Feladat: Tervezzünk egy saját sztring osztályt és készítsünk hozzá néhány operátort! A példában a *stringS* objektumok közösen használják az azonos karaktersorozatokat. (*MINTA11_04*)

```
//OSZTALY.H
#include <iostream>
using namespace std;
#include <cstring>

class stringS
{
  private:
    struct st
    {
      char *ps; // karakterre mutató pointer
      int van;  // jelzi, hogy volt-e értékadás
    };
    st *str;    // st struktúrára mutat
  public:
    stringS();
    stringS(char *);
    stringS(stringS&);
    ~stringS();
    // operátorok túlterhelése: << >> == !=
    friend ostream& operator << (ostream&, stringS&);
    friend istream& operator >> (istream&, stringS&);
    friend int operator == (stringS &s, char *sc)
    { return strcmp(s.str->ps, sc)== 0; }
    friend int operator != (stringS &s, char *sc)
    { return strcmp(s.str->ps, sc)!= 0; }
    friend int operator == (stringS &s1, stringS &s2)
    { return strcmp(s1.str->ps, s2.str->ps)== 0; }
    friend int operator != (stringS &s1, stringS &s2)
    { return strcmp(s1.str->ps, s2.str->ps)!= 0; }
};
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include <cstring>
#include "Osztaly.h"

stringS::stringS() // A paraméter nélküli konstruktor.
{
  str = new st;
  str->ps = 0;
  str->van = 1; // A van jelzi, hogy volt-e értékadás.
}
```

```
stringS::stringS(char *sz) // Sztring argumentum (char *) átvétele
{ str = new st;
  str->ps = new char[strlen(sz)+1];
  strcpy(str->ps, sz);
  str->van = 1;
}

stringS::stringS(stringS& r) // Referenciával adott sztring átvétele
{ r.str->van++;
  str = r.str;
}
// Destruktor, amely törli a struktúra ps adattagját, aztán a
// struktúra pointerét, addig míg a van értéke nem lesz 0.
stringS::~stringS()
{
  if (--str->van == 0)
  { delete str->ps;  delete str; }
}
// Kiírás (<<) operátorának túlterhelése.
ostream& operator << (ostream& sz, stringS& s)
{  return sz << s.str->ps; }
// Olvasás (>>) operátorának túlterhelése.
istream& operator >> (istream& sz, stringS& s)
{ char sbuff[512];
  sz >> sbuff;
  s = sbuff;
  cout << "string: " << s << endl;
  return sz;
}
```

```
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
int main()
{ stringS s1;
  stringS s2("Meredek");
  stringS s3 = s2;
  cout << "Karaktersorozat: ";
  cin >> s1;             // sztring beolvasása
  cout << s1 << endl; // sztring kiíratása
  if (s1 == s2 )         // két sztring azonosságának vizsgálata
    cout << s1 << " azonos " << s2 << endl;
  else
    cout << s1 << " nem azonos " << s2 << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Karaktersorozat: szalma
string: szalma
szalma
szalma nem azonos Meredek
```

Feladatok

1. Használjuk az alábbi definíciót!(GYAK11_1)

```
//OSZTALY.H
class KarakterTomb
{
  private:
    char Buffer[1024];
  public:
    KarakterTomb(char *s)
    {
      strcpy(Buffer,s);
    }
    char* operator +(char* s);
    char* operator -(char betu);
    void TombKiir();
};
```

```
//MAIN.CPP

int main()
{
  KarakterTomb str("valami");
  str = str+ " fontos";
  str.TombKiir();
  str = str - '_';
  str.TombKiir();
  cin.get();
  return 0;
}
```

Egy 1024 bájt nagyságú karaktertömbhöz készítsük el + operator-túlterhelést, amely a tömbhöz hozzáadja a argumentumlistán szereplő karaktertömböt az ***strcat()*** függvénnyel! Írjunk egy – operátor-túlterhelést, amely az adott karaktert a magánhangzók mögé szúrja be.

```
char* KarakterTomb::operator -(char betu)
{
  char s[1024]; int i,j = -1;
  for (i = 0; i<strlen(Buffer); i++)
  {
    switch (Buffer[i])
    {
     case 'a':
     case 'e':
     case 'u':
     case 'i':
     case 'o':
           s[++j] = Buffer[i]; s[++j] = betu; break;
     default: s[++j]= Buffer[i];
   }
  }
  s[++j] = '\0';
  return strcpy(Buffer, s);
}
```

```
char* KarakterTomb::operator +(char* s)
{ return strcat(Buffer,s); }
void KarakterTomb::TombKiir()
{ cout << "Tartalom: " << Buffer << endl;}
```

A program futásának eredménye:

```
Tartalom: valami fontos
Tartalom: va_la_mi_ fo_nto_s
```

2. Használjuk az alábbi definíciót! (GYAK11_2)

```
class matrix
{ private:
    double x[3][3];
  public:
    matrix();
    friend matrix operator +(matrix& a, matrix& b);
    friend matrix operator *(matrix& a, double szor);
    void Kiir();
};
```

Egy 3x3 a szimmetrikus mátrixot soronként töltsünk fel: 1,2,3 4,5,6 7,8,9 adatokkal! Írjunk **friend** függvényeket a két mátrix összeadására, és a mátrix elemeinek egy adott értékkel való szorzására! Segítség a GYAK11_2 feladathoz:

```
//OSZTALY.CPP
matrix::matrix()
{ double k= 1;
  for (int i = 0; i<3; i++)
  {
    for (int j = 0; j<3; j++)
        x[i][j] = k+j;
    k = k+3;
   }
}
matrix operator +(matrix& a, matrix& b)
{ matrix ossz;
  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      ossz.x[i][j] = a.x[i][j] + b.x[i][j];
  return ossz;
}
```

```
//MAIN.CPP
int main()
{
   matrix  x, y, z, w;
   y.Kiir();
   z.Kiir();
   z = x + y;
   z.Kiir();
   w = z * 3;
   w.Kiir();
   cin.get();
   return 0;
}
```

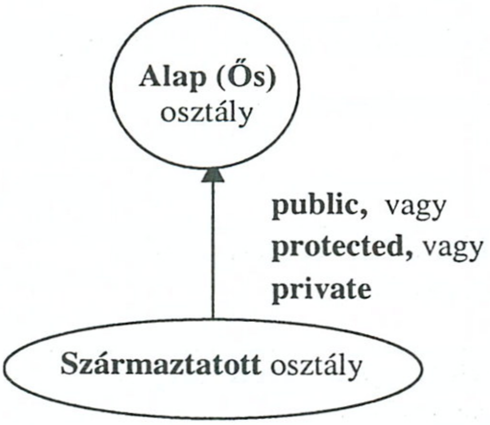
A program futásának eredménye:

```
 matrix tartalma
 1       2       3
 4       5       6
 7       8       9
  matrix tartalma
 1       2       3
 4       5       6
 7       8       9
  matrix tartalma
 2       4       6
 8      10      12
14      16      18
  matrix tartalma
 6      12      18
24      30      36
42      48      54
```

12. Gyakorlat

Rövid elméleti összefoglaló

- **Az öröklés mechanizmusa:**
  - Az öröklés (*inheritance*) azt jelenti, hogy a meglévő osztályokból származtatott újabb osztályok öröklik a definiálásukhoz használt alaposztályok adatstruktúráit és függvényeit, ugyanakkor újabb tagokat is definiálhatnak, vagy régi tagfüggvényeket újraértelmezhetnek.
  - Származtatott osztály (*derived class*) olyan osztály, amely az adattagjait és tagfüggvényeit egy vagy több előzőleg definiált osztálytól örökli. Azt az osztályt, amelytől a származtatott osztály közvetlenül örököl, alaposztálynak (*base class*) nevezzük. A származtatott osztály szintén lehet alaposztálya további osztályoknak, lehetővé téve ezzel az osztály-hierarchia kialakítását.
  - Ha a származtatott osztály ugyanolyan néven és típussal definiálja az alaposztály tagfüggvényeit, akkor tetszőlegesen megváltoztathatja azok algoritmusát.
  - A C++ támogatja a többszörös öröklést (*multiple inheritance*), melynek folyamán az új osztályt több alaposztályból származtatjuk.
  - A származtatás szintaxisa a származtatandó osztály neve után kettőspont (:) majd a származtatási mód (**public**, **private**, **protected**) az osztály nevével. Többszörös öröklés esetén vesszővel elválasztva újabb osztályokat csatlakoztathatunk a származtatáshoz.



A **Származtatott** osztály mindent lát az **Alap** osztályból, kivéve annak **private** tagjait.

```
class Alap
{
  private:
    int sajat;
  protected:
    int orokolheto;
  public:
    int nyilvanos;
};
```

```
                      ⎧private        ⎫
class Szarmaztatott : ⎨protected Alap ⎬
                      ⎩public         ⎭
{ };
```

```
Szarmaztatott use;    // a származtatott osztály felhasználója,
                      //"példányosítója"

// a származtatott osztály továbbfejlesztője, örököse
class Dev : public Szarmaztatott
{ };
```

A kérdés, hogy mit érhet el az *Alap* osztályból a *Szarmaztatott* osztály felhasználója, illetve a továbbfejlesztője:

Alap osztálybeli hozzáférés	Származtatási mód	*Szarmaztatott* osztálybeli hozzáférés	Mit lát a *Szarmaztatott* osztály felhasználója az *Alap*-ból?
private **protected** **public**	**private** (alapértelmezett)	– **private** **private**	semmit
private **protected** **public**	**protected**	– **protected** **protected**	semmit
private **protected** **public**	**public**	– **protected** **public**	Az *Alap* **public** tagjait (*nyilvános*)

Alap osztálybeli hozzáférés	Származtatási mód	Mit lát a *Szarmaztatott* osztály továbbfejlesztője az *Alap*-ból?
private **protected** **public**	**private** (alapértelmezett)	semmit
private **protected** **public**	**protected**	Az *Alap* **public** és **protected** tagjait (*nyilvános*, *örökölhető*)
private **protected** **public**	**public**	Az *Alap* **public** és **protected** tagjait (*nyilvános*, *örökölhető*)

```
//MUV.H                                                    (MINTA12_01)
class Muvelet
{ protected:
      double a,b,ered; // a,b az adat, ered az eredmény
  private:
      double hatvany;
  public:
      Muvelet(double a1, double b1);
      void Szamol();
      void Kiir();
      void Hatvanyoz();
      double GetHatvany(){ return hatvany;}
  private:
      void Hibajelzes();
};
```

```
class UjMuvelet : public Muvelet
{
  protected:
    int jel;
  public:
    UjMuvelet(double a1, double b1, int jel1);
    void Szamol();
    void Kiir();
};
```

Magyarázat a feladathoz:
Muvelet osztály
protected adattagjai:
- *a,b* adattagok bemenő adatok, ezeknek adunk kívülről kezdőértéket,
- *ered* tárolja az eredményt.

private adattagja, ehhez a származtatott osztály nem fér hozzá közvetlenül:
- *hatvany*

public tagfüggvények:
- konstruktor,
- *Szamol()* számtani közepet számol,
- *Kiir()* kiírja az eredményt,
- *Hatvanyoz()* az *a* adattagot emeli a *b* adattagra,
- *GetHatvany()* a hatványozás eredményét adja vissza.

private tagfüggvény:
- *Hibajelzes()* a bemenő adatok csak pozitív számok lehetnek, a tagfüggvény a negatív értékű adat abszolút értékét veszi.

UjMuvelet osztály publikusan származik a *Muvelet* osztálytól:
protected saját adattagja:
- *jel* 0 esetén az $(a \cdot b)^2$,
  1 esetén a $\sqrt{a \cdot b}$.

Szamol() és a *Kiir()* tagfüggvények az alaposztály hasonló nevű tagfüggvényeit elfedik.

```
//MUV.CPP
#include <iostream>
#include <cmath>
using namespace std;
#include "muv.h"
Muvelet::Muvelet(double a1, double b1)
{
  a = a1; b = b1; Hibajelzes();
}
void Muvelet::Szamol()
{ // számtani közepet számol
  ered = (a+b)/2;
}
void Muvelet::Kiir()
{
  cout <<"Muvelet: eredmeny: " << ered << endl;
}
```

```
void Muvelet::Hatvanyoz()
{  hatvany = pow(a,b); }
void Muvelet::Hibajelzes()
{
   if (a < 0)
   {
     cerr << "Az 1. adat negativ, "
     << "veszi az abszolut erteket! " << endl;
      a = fabs(a);
   }
   if (b < 0)
   {
     cerr << "A  2. adat negativ, "
     << "veszi az abszolut erteket! " << endl;
      b = fabs(b);
   }
}

UjMuvelet::UjMuvelet(double a1, double b1, int jel1):Muvelet(a1,b1)
{ jel = jel1; }

void UjMuvelet::Szamol()
{
  // Hibajelzes(); nem hívható, mert private
  if (jel)
    ered = sqrt(a*b); // mértani középet számol
  else
    ered = pow(a*b,2); // a szorzat négyzetét számolja
 //hatvany=0; nem férhető hozzá, mivel private
}
void UjMuvelet::Kiir()
{
 cout << "UjMuvelet: " << jel
      << "  eredmeny: " << ered << endl;
}
```

Magyarázat:

Muvelet osztály tagfüggvényei:

A konstruktor értéket ad az *a,b* adattagoknak, és aktiválja a *Hibajelzes()* tagfüggvényt a bemenő adatok vizsgálatára.

A *Szamol()* tagfüggvény az *a,b* adattagok számtani közepét számítja ki az *ered* adattagba.

A *Kiir()* tagfüggvény kiírja, hogy melyik osztályhoz tartozik az eredmény.

A *Hatvanyoz()* tagfüggvény az *a* adattagot emeli a *b* adattagra, és a *hatvany* adattagba tárolja az eredményt.

A *Hibajelzes()* tagfüggvény a negatív adattagok előjelét megfordítja. Ezt a függvényt a származtatott osztály nem éri el, mivel **private** hozzáférésű.

UjMuvelet osztály tagfüggvényei:

Az *UjMuvelet* konstruktora *a1,b1* paraméterét továbbadja a *Muvelet* konstruktornak, a saját adattagját a függvény törzsében inicializálja.

A *Szamol()* tagfüggvény a *jel* értékétől függően számol.
1 esetén, az *a* és *b* mértani közepét számítja ki az *ered* adattagba,
0 esetén az *a* és *b* szorzatának négyzetét számítja ki az *ered* adattagba.

Kiir() tagfüggvény jelzi az osztály nevét, és kiírja az eredményt.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "muv.h"
int main()
{ double x,y;
  cout << "1. adat: "; cin >> x;
  cout << "2. adat: "; cin >> y;
  Muvelet w1(x,y);
  w1.Szamol(); // számtani közepet számol
  w1.Kiir();
  w1.Hatvanyoz();
  cout << "Hatvany erteke: " << w1.GetHatvany()
  << endl << endl;

  UjMuvelet w2(x,y,1);
  w2.Szamol();  // mértani közepet számol
  w2.Kiir();
  w2.Hatvanyoz();
  cout << "Hatvany erteke: " << w2.GetHatvany()
  << endl << endl;
  UjMuvelet w3(x,y,0);
  w3.Szamol();  // szorzat négyzetét számolja
  w3.Kiir();
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
1. adat: 2
2. adat: 3
Muvelet: eredmeny: 2.5
Hatvany erteke: 8

UjMuvelet: 1  eredmeny: 2.44949
Hatvany erteke: 8

UjMuvelet: 0  eredmeny: 36
```

- Egy alaposztály "barátja" (**friend**) a származtatott osztályban csak az alaposztálytól örökölt tagokat érheti el. Egy származtatott osztály "barátja" (**friend**) az alaposztályból csak a **public** és a **protected** tagokat érheti el.

- **Többszintű öröklés bemutatása családfán**

A nagypapa, a szülő és az unoka családfán mutatjuk be a **public**, **protected** és **private** adattagok elérését a különböző öröklési hozzáférések esetén. (*MINTA12_02*)

```
//OSZTALY.H
class nagypapa
{
  public:
    int konyvtar;
  protected:
    int kert;
  private:
    int album;
  public:
  nagypapa();
  void Kiir();
};

class szulo: public nagypapa
{
  public:
    int fenykepezogep;
  protected:
    int lakas;
  private:
    int szamitogep;
  public:
  szulo();
  void Kiir();
  void Hozzafer();
  void NemFerHozza();
};
```

```
class unoka :public szulo
{
  public:
    int jatek;
  protected:
    int szemelyauto;
  private:
    int vitorlas;
  public:
  unoka();
  void Kiir();
  void Hozzafer();
  void NemFerHozza();
};
```

```
//OSZTALY.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;

nagypapa::nagypapa()
{ konyvtar = 1;
  kert = 2;
  album = 3;
}

void nagypapa::Kiir()
{ cout << "nagypapa tulajdona"  << endl;
  cout << "public konyvtar: " << konyvtar << endl;
  cout << "protected kert: " << kert << endl;
  cout << "private album: " << album << endl;
}
```

```
szulo::szulo()
{ fenykepezogep =4 ;
  lakas = 5;
  szamitogep = 6;
}
void szulo::Kiir()
{ cout << endl << "szulo tulajdona" << endl;
  cout << "public fenykepezogep: "<< fenykepezogep << endl;
  cout << "protected lakas: " << lakas << endl;
  cout << "private szamitogep: " << szamitogep << endl;
}
void szulo::Hozzafer()
{ cout << "\nOROKLI: nagypapatol "  << endl;
  cout << "public nagypapa: konyvtar: "<< konyvtar << endl;
  cout << "protected nagypapa: kert: " << kert << endl;
}
void szulo::NemFerHozza()
{ cout << "\nNEM OROKLI "  << endl;
  cout <<"private nagypapa: album: "<<" nem fer hozza!!"<< endl;
}

unoka::unoka()
{ jatek = 7;
  szemelyauto = 8;
  vitorlas = 9;
}
void unoka::Kiir()
{ cout << endl << "unoka tulajdona" << endl;
  cout << "public jatek: " << jatek << endl;
  cout << "protected szemelyauto: " << szemelyauto << endl;
  cout << "private vitorlas: " << vitorlas << endl;
}
void unoka::Hozzafer()
{ cout << "\nOROKLI: nagypapatol "  << endl;
  cout << "public  nagypapa: konyvtar: "<< konyvtar<< endl;
  cout << "protected nagypapa: kert: " << kert << endl;
  cout << "\nOROKLI: szulotol" << endl;
  cout << "public  szulo: fenykepezogep: "<<fenykepezogep
       << endl;
  cout << "protected szulo: lakas: " << lakas << endl;
}
void unoka::NemFerHozza()
{ cout << "\nNEM OROKLI "  << endl;
  cout << "private nagypapa: album: " <<"nem fer hozza!!"
       << endl;
  cout << "private szulo: szamitogep: "<< "nem fer hozza!!"
       << endl;
}
```

```
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;
```

```
int main()
{
  nagypapa n;
  szulo sz;
  unoka u;
  cout << "szulo public oroklese nagypapatol\n";
  n.Kiir();
  cout <<"\nEnter -> tovabb";
  cin.get();
  sz.Kiir(); sz.Hozzafer(); sz.NemFerHozza();
  cout <<"\nEnter -> tovabb";
  cin.get();
  u.Kiir(); u.Hozzafer(); u.NemFerHozza();
  cin.get();
  return 0;
}
```

A *MINTA12_02* program futásának eredménye **public** öröklés esetén:

```
szulo public oroklese nagypapatol

nagypapa tulajdona
public konyvtar: 1
protected kert: 2
private album: 3
szulo tulajdona
public fenykepezogep: 4
protected lakas: 5
private szamitogep: 6
OROKOLI: nagypapatol
public nagypapa: konyvtar: 1
protected nagypapa: kert: 2
NEM OROKLI
private nagypapa: album: nem fer hozza!!

unoka tulajdona
public jatek: 7
protected szemelyauto: 8
private vitorlas: 9
OROKLI: nagypapatol
public    nagypapa: konyvtar: 1
protected nagypapa: kert: 2
OROKLI: szulotol
public    szulo: fenykepezogep: 4
protected szulo: lakas: 5
NEM OROKLI
private nagypapa: album: nem fer hozza!!
private szulo:    szamitogep: nem fer hozza!!
```

A *MINTA12_03* program futásának eredménye **protected** öröklés esetén.

```
unoka tulajdona
public jatek: 7
protected szemelyauto: 8
private vitorlas: 9
OROKLI: nagypapatol
public -> protected nagypapa: konyvtar: 1
protected nagypapa: kert: 2
OROKLI: szulotol
public    szulo: fenykepezogep: 4
protected szulo: lakas: 5
NEM OROKLI
private nagypapa: album: nem fer hozza!!
private szulo: szamitogep: nem fer hozza!!
```

A *MINTA12_04* program a **private** öröklést mutatja be, amikor a **szulo** letiltja a **nagypapa** osztálytól való öröklést az **unoka** számára.

Táblázatban összefoglalva látjuk a három mintafeladat eredményét, amely szemlélteti az öröklési hozzáférést.

Osztályok	Saját tagadatai	szulo: **public** nagypapa öröklés hatása *MINTA12_02*	szulo: **protected** nagypapa öröklés hatása *MINTA12_03*
nagypapa	**public**: konyvtar **protected**:kert **private**: album		
szulo	**public**: fenykepezogep **protected**:lakas **private**: szamitogep	**public** konyvtar **protected** kert	**public** konyvtar **protected** kert
öröklési hozzáférés hatása		↓ nem változik	↓ **public** → **protected**
unoka	**public**: jatek **protected**:szemelyauto **private**: vitorlas	**public** konyvtar **protected** kert --------------- **public** fenykepezogep **protected** lakas	**protected** konyvtar **protected** kert ------------------ **public** fenykepezogep **protected** lakas

Osztályok	Saját tagadatai	szulo: **private** nagypapa öröklés hatása *MINTA12_04*
nagypapa	**public**: konyvtar **protected**:kert **private**: album	
szulo	**public**: fenykepezogep **protected**:lakas **private**: szamitogep	**public** konyvtar **protected** kert
öröklési hozzáférés hatása		↓ **public** → **private** **protected** → **private**
unoka	**public**: jatek **protected**:szemelyauto **private**: vitorlas	**private** - nincs öröklés ------------------------- **public** fenykepezogep **protectec** lakas

A *main*() függvényben az objektumpéldányok a származtatott osztályból **protected** és **private** öröklés esetében már nem érik el az alaposztály **public** adattagjait és tagfüggvényeit sem az öröklés hatására. (*MINTA12_05*)

```
//OSZTALY.H
class nagypapa
{
  public:
    int konyvtar;
    . . .
 };
class szulo : protected nagypapa
{
  public:
   int fenykepezogep;
   . . .
class unoka :private szulo
{
  public:
    int jatek;
    . . .
 };
```

```
//MAIN.CPP
#include <iostream>
#include "Osztaly.h"
using namespace std;

int main()
{
  szulo sz;
  unoka u;
  cout << "nagypapa konyvtara"
       << "szulon keresztul\n";
  cout << sz.konyvtar << endl;
  cout << "szulo fenykepezogepe"
       << "unokan keresztul\n";
  cout << u.fenykepezogep << endl;
  cin.get();
  return 0;
}
```

A *MINTA12_05* program fordításakor hibát kapunk a **class** szulo : **protected** nagypapa öröklés miatt, mert ez a külvilág felé az adattag **public** hozzáférését hozzáférhetetlenné teszi, mivel **public**-ot **protected**-re módosítja. A főprogramból nem érjük el a *szulo sz* (nagypapától örökölt) könyvtárat.

Ugyanígy van az *unoka u* objektumpéldánya esetén, ott a **private** hozzáférés okozza azt, hogy nem érjük el a *szulo* osztály **public** *fenykepezogep* adattagját, mivel **private** típusúvá válik.

A fordításban az *inaccessible* jelenti azt, hogy hozzáférhetetlenek lettek a **public** adattagok a mintafeladatban.

- **Több osztálytól való öröklés** esetén a szabály a következő: az első osztálytól való örökléskor : (kettőspontot használunk), a továbbiaknál pedig , (vesszővel) kell felsorolni az osztályokat, előtte jelezni a hozzáférést (**public**, **protected**, **private**).
  Pl. `class Osztaly3 : public Osztaly2, public Osztaly1`

Több osztálytól való öröklés bemutatása. *(MINTA12_06)*

```
//OSZTALY.H
class Adatok
{
  protected:
     double a,b;
  public:
    Adatok(double a1, double b1)
    { a = a1; b = b1; }
    ~Adatok(){};
};
```

```
class Muvelet
{
  protected:
    int muvjel;
  public:
    Muvelet(int muvjel1)
      { muvjel = muvjel1;}
    ~Muvelet(){};
};
class Vegrehajt : public Adatok, public Muvelet
{
   protected:
     double eredmeny;
   public:
     Vegrehajt( double a1, double b1, int muvjel1);
     ~Vegrehajt(){};
     void Szamol();
     void SetMuvjel(int muvjel1)
     {
       muvjel = muvjel1; Szamol();
     }
     void SetAdatok(double a1, double b1)
     { a = a1; b = b1;
       Szamol();
     }
     double GetEred() { return eredmeny;}
     void Kiir();
};
```

Adatok osztály:	*a,b* két (valós) adattagja van, melyeknek a konstruktor ad értéket.
Muvelet osztály:	*muvjel* (egész) adattagja van, melynek a konstruktor ad értéket.
Vegrehajt osztály:	*eredmény* (valós) adattagja van, a többi adattagot az *Adatok* és a *Muvelet* osztálytól öröklés utján éri el. **void** `Vegrehajt::Szamol()` tagfüggvény a *muvjel* értéke alapján végez számítást az *eredmeny* adattagba. A *muvjel* értéke lehet: 0 akkor az *a* és a *b* számtani közepét, 1 esetén az *a* és a *b* mértani közepét számítja ki. **void** `Vegrehajt::Kiir()` tagfüggvény az adattagokat és az eredményt írja ki, a műveletet pedig szövegesen jelzi.

```
//OSZTALY.CPP
#include "Osztaly.h"
#include <cmath>
#include <iostream>
using namespace std;

Vegrehajt::Vegrehajt(double a1,double b1, int muvjel1):Adatok(a1,b1),
                     Muvelet(muvjel1)
{
    eredmeny = 0;
}
```

```
void Vegrehajt::Szamol()
{ switch (muvjel)
  {
    case 0: eredmeny = (a+b)/2; break;
    case 1: eredmeny = sqrt(a*b); break;
    default: cerr << "Hibas a muvelet\n!";
  }
}
void Vegrehajt::Kiir()
{ int hiba = 1;
  switch (muvjel)
  {
    case 0: cout << "\nSzamtani kozep szamitasa" << endl;
            hiba = 0; break;
    case 1: cout << "\nMertani kozep szamitasa" << endl;
            hiba = 0; break;
    default: cerr << "Hibas a muveleti jel \n";
  }
  if (hiba == 0)
  { cout <<"1.adat: " << a << "\n2.adat: " << b
         << "\neredmeny: " << eredmeny << endl;
  }
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{ Vegrehajt v(1,2,0);
  v.Szamol();
  cout <<"Eredmeny: "<< v.GetEred()<<endl;
  v.SetMuvjel(1);
  cout <<"\nEredmeny: " << v.GetEred()
       << endl;
  v.SetAdatok(2,4);
  v.Kiir();
  v.SetMuvjel(0);
  v.Kiir();
  v.SetMuvjel(1);
  v.Kiir();
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Eredmeny: 1.5

Eredmeny: 1.41421

Mertani kozep szamitasa
1.adat: 2
2.adat: 4
eredmeny: 2.82843

Szamtani kozep szamitasa
1.adat: 2
2.adat: 4
eredmeny: 3

Mertani kozep szamitasa
1.adat: 2
2.adat: 4
eredmeny: 2.82843
```

Feladatok

1. Használjuk az alábbi definíciókat!

```
class Teglalap
{
  protected:
    double a_oldal, b_oldal;
    double ered1, ered2;
  public:
     Teglalap(double a1, double a2);
     void Szamol1();  // kerületet számol az ered1 tagadatba
     void Szamol2();  // területet számol az ered2 tagadatba
     double GetEred1(){ return ered1; }
     double GetEred2(){ return ered2; }
};
class Teglatest : public Teglalap
{
  protected:
    double c_oldal;
  public:
     Teglatest(double a1, double b1, double c1 );
     void Szamol1();  // felszínt számol az ered1 tagadatba
     void Szamol2();  // térfogatot számol az ered2 tagadatba
};
```

Írjuk meg a tagfüggvényeket, és a ***main***() függvényt! Az objektumpéldányt inicializáljuk konstruktorral, és írassuk ki az eredményeket! (GYAK12_1)

2. Tervezzünk egy *Negyzet* osztályt, és származtassunk belőle egy *Kocka* osztályt!

```
class negyzet
{
  protected:
     double a_oldal;
     double ered1, ered2;
  private:
     double atlo;
  public:
     negyzet(double a1);
     void Szamol1();  // kerület számítása
     void Szamol2();  // terület számítása
     double GetEred1() { return ered1; }
     double GetEred2() { return ered2; }
     friend double AtlotSzamol(negyzet& n);
};
class kocka : public negyzet
{
  private:
     double testatlo;
  public:
     kocka(double a1);
     void Szamol1();  // felszín számítása
     void Szamol2();  // térfogat számítása
     friend double TestatlotSzamol(kocka& k);
};
```

Tervezzünk hozzá tagfüggvényeket és ***main***() függvényt, az eredményeket pedig jelenítsük meg! GYAK12_2)

13. Gyakorlat

- **Öröklési példa a tagfüggvény újradefiniálására, mely további vizsgálatok alapjául szolgál.**

A tagfüggvény újradefiniálásával szorzat helyett hányados számítása. (*MINTA13_01*)

```
//OSZTALY.H
class Alap
{
   protected:
     // x,y adattag az adat, az ered tárolja az eredményt
     double x,y, ered;
   public:
     Alap(double x1, double y1);
     void Szamol();
     void Kiir();
};
// A Szarmaztatott osztály a Szamol() tagfüggvényt újradefiniálja.
class Szarmaztatott : public Alap
{
   public:
     Szarmaztatott(double x1, double y1);
     void Szamol();
};
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
// Alap osztály konstruktora kezdőértéket ad az x,y adattagoknak,
// és az ered tartalmát nullázza.
Alap::Alap(double x1, double y1)
{
  x = x1; y = y1; ered = 0;
}
// A két adat (x,y) szorzatát számítja ki az ered adattagba.
void Alap::Szamol()
{
   ered = x*y;
}
// Az ered tartalmát írja ki.
void Alap::Kiir()
{ cout << "Eredmeny: " << ered << endl; }

Szarmaztatott::Szarmaztatott(double x1, double y1)
:Alap(x1, y1)
{ }
// A két adat (x,y) hányadosát számítja ki az ered adattagba.
void Szarmaztatott::Szamol()
{
  ered = x / y;
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"

int main()
{
  double a = 5, b = 2;
  Alap p(a,b);
  p.Szamol();  // szorzat számítása
  p.Kiir();
  Szarmaztatott sz(a,b);
  sz.Szamol();  // hányados számítása
  sz.Kiir();
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Eredmeny: 10
Eredmeny: 2.5
```

- **Objektumok zártsága, early binding (korai kötés)**
  Ha az alaposztály tagfüggvénye másik, saját tagfüggvényét aktiválja, és a származtatott osztály az alaposztálynak ezt tagfüggvényét hívja, akkor nem várt hatású eredmény keletkezik. A jelenség oka az *objektumok zártsága*. A zártság elvének érvényesüléséről a fordítóprogram azzal gondoskodik, hogy a tagfüggvényekből elvégzett tagfüggvényhívásokat a hagyományos függvényhívásokkal azonos módon (rögzített címmel) fordítja. Ebben az esetben az alaposztály és nem a származtatott osztály tagfüggvénye hívódik meg. A fordítás során megvalósított összerendelést korai kötésnek (**early binding**) nevezzük.

A *MINTA13_02* a *MINTA13_01* mintafeladat módosítása. Az *Alap* osztály bővült egy *Vegrehajt()* tagfüggvénnyel, amely a *Szamol()* tagfüggvényt aktiválja.

```
//OSZTALY.H

class Alap
{
   protected:
     double x,y, ered;
   public:
     Alap(double x1, double y1);
     void Szamol();
     void Kiir();
     void Vegrehajt(){ Szamol(); }
};
class Szarmaztatott  : public Alap
{
   public:
     Szarmaztatott(double x1, double y1);
     void Szamol();
};
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
```

```
int main()
{ double a = 5, b = 2;
  Alap p(a,b);
  p.Vegrehajt(); p.Kiir();
  Szarmaztatott sz(a,b);
  sz.Vegrehajt(); sz.Kiir();
  cin.get();
  return 0;
}
```

Mindkét objektumpéldány a *Vegrehajt()* tagfüggvényt aktiválja, az eredmény, mindkét esetben azonos:

A program futásának eredménye:

```
Eredmeny: 10
Eredmeny: 10
```

Megjegyzés: A korai kötés eredménye, hogy az *Alap* osztály *Szamol()* tagfüggvénye hívódik meg mindkét esetben.

- **Sokalakúság (polymorphism) - virtuális tagfüggvények, late binding (késői kötés)**
  - Az öröklés során az objektumok viselkedése módosulhat, azaz például egy öröklött tagfüggvény neve ugyan nem változik egy leszármazottban, de esetleg már egy kicsit (vagy éppen nagyon) másképp viselkedik. Ezt a C++ nyelvben az ún. virtuális függvények (**virtual functions**) teszik lehetővé.
  - A sokalakúság akkor érvényesül, ha a virtuális tagfüggvényeket használunk. A virtuális tagfüggvények biztosítják, hogy egy adott osztály-hierarchiában (származási fán) egy adott függvény különböző verziói létezzenek úgy, hogy csak a kész program futása során derül ki, hogy ezek közül éppen melyiket kell végrehajtani. Ezt a mechanizmust, azaz a hívó és a hívott függvény futási időben történő összerendelését késői kötésnek (**late binding**) nevezzük.
  - A konstruktor kivételével minden tagfüggvény (beleértve a destruktorokat is) virtuálissá tehető a **virtual** kulcsszó megadásával.
  - Ha egy függvényt az alaposztályban virtuálisként deklarálunk, akkor ezt a tulajdonságát az öröklés során is megőrzi. A származtatott osztályban a virtuális függvényt saját változattal újradefiniálhatjuk, de az öröklött verziót is használhatjuk. Saját verzió definiálásakor nem szükséges a **virtual** kulcsszót megadnunk.

Virtuális tagfüggvény használata. (*MINTA13_03*)

```
//OSZTALY.H
class Alap
{
  protected:
    double x,y, ered;
  public:
    Alap(double x1, double y1);
    virtual void Szamol(); //módosítás: a Szamol() vituális
    void Kiir();
    void Vegrehajt(){ Szamol(); }
};
```

```
class Szarmaztatott : public Alap
{
 public:
  Szarmaztatott(double x1, double y1);
  void Szamol();
};
```

A *MINTA13_03* mintafeladatban a *MINTA13_02* feladathoz képest a módosítás csak a **virtual** kulcsszó használatában van.
Az *Alap* osztályban a

virtual void Szamol();

teszi lehetővé, hogy a *Szarmaztatott* osztály örökölt *Vegrehajt()* tagfüggvényének hívásakor a saját *Szamol()* tagfüggvénye hajtódik végre, így már jó eredményt kapunk.

A *Szarmaztatott* osztályban már **nem kell** a **virtual** kulcsszót kitenni a *Szamol()* tagfüggvény előtt!

A program futásának eredménye:

```
Eredmeny: 10
Eredmeny: 2.5
```

- **Absztrakt osztály használata**
  - Az absztrakt tagfüggvénynek tiszta virtuális (**pure virtual**) függvénynek kell lennie.
  - A **virtual** lehetővé teszi, hogy öröklés során felülírható legyen.
  - Az absztrakt osztály metódusmutatója NULL, tehát nem tartozik hozzá kód.
  - Az absztrakt osztály nem használható objektumok létrehozására.
  - Az absztrakt osztály használatának célja, hogy minden utódosztály az örökölt virtuális metódusokat a feladatnak megfelelően felülírják.

Absztrakt osztály használatának bemutatása. (*MINTA13_04*)

```
//OSZTALY.H
class Alakzat  // absztrakt osztály
{
   protected:
    virtual void Szamol1() = 0; // tiszta virtuális,
                                // kód nélküli metódusok.
    virtual void Szamol2() = 0;
    virtual void Kiir() = 0;
};
// A Teglalap osztály az Alakzat osztálytól örökölt metódusokat
// írja felül.
class Teglalap : public Alakzat
{
  protected:
     double a,b, ered1,ered2; // a, b  -  téglalap két oldala
     // ered1 a kerület, ered2 a terület
  public:
     Teglalap(double a1, double b1);
     void Szamol1();
     void Szamol2();
     void Kiir();
};
```

```
// A Kor osztály szintén az Alakzat osztálytól örökölt metódusokat
// írja felül.
class Kor : public Alakzat
{
  protected:
     double r, ered1, ered2; // r a kör sugara
     // ered1 a kerület, ered2 a terület
  public:
     Kor(double r1);
     void Szamol1();
     void Szamol2();
     void Kiir();
};
```

```
//OSZTALY.CPP
#include "Osztaly.h"
#include <iostream>
using namespace std;
const double PI = 3.141592654;
Teglalap::Teglalap(double a1, double b1)
{
   a = a1; b = b1; ered1 = 0; ered2 = 0;
}
void Teglalap::Szamol1()
{ ered1 = a*b; }         // a téglalap területét számítja
void Teglalap::Szamol2()
{ ered2 = 2*(a + b); } // a téglalap kerületét számítja
void Teglalap::Kiir()
{
  cout << endl << "A tegalalap terulete: " << ered1 << endl;
  cout << "A tegalalap kerulete: " << ered2 << endl;
}
Kor::Kor(double r1)
{ r = r1; ered1 = 0; ered2 = 0; }
void Kor::Szamol1()
{ ered1 = r * r * PI; } // a kör területét számítja
void Kor::Szamol2()
{ ered2 = 2 * r * PI; } // a kör kerületét számítja
// A Kiir() tagfüggvény az eredményeket jeleníti meg.
void Kor::Kiir()
{
  cout << endl << "A kor terulete: " << ered1 << endl;
  cout << "A kor kerulete: " << ered2 << endl;
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
```

```
int main()
{
  Teglalap t(1,2);
  cout << "Absztrakt osztaly hasznalata"
       << endl;
  t.Szamol1(); t.Szamol2(); t.Kiir();
  Kor k(1);
  k.Szamol1(); k.Szamol2(); k.Kiir();
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Absztrakt osztaly hasznalata

A tegalalap terulete: 2
A tegalalap kerulete: 6

A kor terulete: 3.14159
A kor kerulete: 6.28319
```

- **Statikus adattagok használata**
  - Ha egy adattagot a **static** kulcsszóval deklarálunk, akkor *ez csak egy példányban létezik*, függetlenül attól, hogy mennyi objektumpéldányt hozunk létre.
  - A statikus adattag inicializálása *nem a konstruktorban* történik.
  - A statikus adattagot a *globális változókhoz hasonlóan a függvényen kívül* definiáljuk.
  - Inicializáláskor a *hatókör operátorral az osztály nevét is meg kell adnunk* az eléréshez
  - Az inicializálás *csak egyszer szerepelhet a programban*, ezért nem lehet az osztály deklarációját tartalmazó include állományban, hanem *csak az implementációs állományban adhatunk neki értéket.*

- **Struktúra használata**

  Struktúra típusdefiníció lehetővé teszi, hogy az osztály dinamikusan létrehozzon egy olyan tömböt, melynek minden eleme a struktúra elemei.

A **static** adattag használatának bemutatása. (*MINTA13_05*)

```
// OSZTALY.H
 struct aru{
             char azonosito[15]; // az áru megnevezése
             int darab;          // az áru darabszáma
             double ara;         // az áru ára
           };
class raktar
{
  protected:
    aru* rak;   // az aru struktúrára mutató pointer.
    static int arufajta; // statikus adattag, az áruféleségek száma
    double osszar;       // az áruk össz ára
    int osszdarab;       // az áruk összes darabszáma
  public:
    raktar();
    ~raktar();
    void Keszlet();
    void Kiir();
};
```

```
// OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
```

```
 raktar::raktar(int arufajta)
 {
    rak = new aru[arufajta];     // helyfoglalás
    cout << "Raktar feltoltese: " << arufajta << " fajta aruval";
    cout << endl << endl ;
    for (int i = 0; i< arufajta; i++)  // adatok beolvasása
    {
      cout << "Aru neve: "; cin >> rak[i].azonosito;
      cout << "Aru db  : "; cin >> rak[i].darab;
      cout << "Aru ara : "; cin >> rak[i].ara;
      cout << endl;
    }
 }
// dinamikusan lefoglalt memória-terület felszabadítása
 raktar::~raktar()
 {  delete [] rak; }
 void raktar::Keszlet()
 { osszar = 0; osszdarab = 0;
   for (int i = 0; i<arufajta; i++)
    {
      osszar += rak[i].darab * rak[i].ara; // az áruk összértéke
      osszdarab += rak[i].darab;           //az áruk össz darabszáma
    }
 }
 void raktar::Kiir()  // az eredmények megjelenítése
 {
    cout << endl <<"Raktarkeszlet" << endl << endl;
    cout << "Osszertek : " << osszar << endl;
    cout << "Osszdarab : " << osszdarab << endl;
 }
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int raktar::arufajta = 3;
int main()
{
  raktar *r;
  r = new raktar;
  r->Keszlet();
  r->Kiir();
  delete r;

  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Raktar feltoltese: 3 fajta aruval

Aru neve: asztal
Aru db  : 2
Aru ara : 1000

Aru neve: szek
Aru db  : 4
Aru ara : 500

Aru neve: szekreny
Aru db  : 1
Aru ara : 2000

Raktarkeszlet

Osszertek : 6000
Osszdarab : 7
```

Megjegyzés: A főfüggvény aktiválja a dinamikus objektumpéldány létrehozására a konstruktort 3 féle árufajtával. Meghívja a *Keszlet()* és a *Kiir()* tagfüggvényeket. A **delete** hívása aktiválja a destruktort, amely felszabadítja az *r* konstruktorában lefoglalt memóriát.

Feladatok

1. Használjuk az alábbi definíciót! (GYAK13_1)

```
//OSZTALY.H
class Kor
{
    protected:
       double sugar;
       double ered1, ered2;

    public:
       Kor(double r);
       virtual void Eredmeny1();                // kör kerülete (2rπ)
       virtual void Eredmeny2();                // kör területe (r^2π)
       // visszaadja az ered1 értékét
       double GetEred1() { return ered2; }
       // visszaadja az ered2 értékét
       double GetEred2() { return ered2; }
       // kiírja szövegesen az eredményeket
       virtual void Kiir();
};

class Gomb : public Kor
{
   public:
      Gomb(double r);
      ~Gomb();
      void Eredmeny1();   // a gömb felszíne (4r^2π)
      void Eredmeny2();   // a gömb térfogata(4r^3π/3)
      void Kiir();        // kiírja szövegesen az eredményeket
};
```

Dolgozzuk ki a tagfüggvényeket, és írjunk hozzá ***main()*** függvényt!

A program futásának eredménye r = 1 esetén:

```
Kor kerulete: 6.28319
Kor terulete: 3.14159
Gomb felszine : 12.5664
Gomb terfogata: 4.18879
```

2. Tervezzünk egy osztályt, amely kiszámítja két vektor (egész típusú dinamikus tömb) skalárszorzatát. A tömb elemeinek számát *meret* adattaggal adjuk meg (használjuk a **static int** meret; definíciót)! Tervezzünk hozzá adattagokat és tagfüggvényeket! (GYAK13_2)

```
//OSZTALY.H
class Vektorok
{
  private:
    double *x,*y;
    static int meret;
    double skszorzat;

  public:
    Vektorok();
    ~Vektorok();
    void SkalarSzorzat();
    double GetSkSzorzat() { return skszorzat;}
};
```

A konstruktorban helyet foglalunk az *x* és az *y* vektorok számára, például:

```
x = new double[meret];
```

A ***main***() függvény előtt a statikus adattag megadása:

```
int Vektorok::meret = 3;
```

A program futásának eredménye:

```
Elemek szama: 3
0. x elem: 1
0. y elem: 1

1. x elem: 2
1. y elem: 2

2. x elem: 3
2. y elem: 3

Skalarszorzat: 14
```

3. Használjuk az alábbi definíciókat! (GYAK13_3)

```
//OSZTALY.H
typedef struct Kocsi{
                      char tipusa[15];
                      int ajtokszama;
                      int darab;
                      double ara;
                     };
class AutoKereskedes
{
  protected:
    Kocsi* szalon;
    int KocsiTipusokSzama;
    double KocsikOsszara;
    int Osszdarab;
  public:
    AutoKereskedes(int db);
    ~AutoKereskedes();
    void KocsiKeszletDarab();
    void KocsikOsszAra();
    void KocsikLista(); // kiírja az eredményeket
};
```

Dolgozzuk ki a tagfüggvényeket! A ***main***() függvényben olvassuk be a kocsik fajtájának darabszámát, és használjunk dinamikus objektumpéldányt! Aktiváljuk a tagfüggvényeket!

```
int main()
{
  int fajta;
  AutoKereskedes *ak;
  . . .
}
//Használjunk fixpontos kiírási módot
 void AutoKereskedes::KocsikLista()
 {
   cout.setf(ios::fixed);
   cout << endl << "Autokereskedes" << endl << endl;
   cout << "Kocsik teljes ara      : " << KocsikOsszara << endl;
   cout << "Kocsik ossz darabszama : " << Osszdarab << endl;
 }
```

A program futásának eredménye:

```
Kocsik fajtajanak darabszama: 3
Autokereskedes feltoltese: 3 fajta kocsitipussal

Kocsi tipusa      : Mercedes
Kocsi ajtok szama: 4
Kocsi db szama    : 2
Kocsi ara         : 3000000

Kocsi tipusa      : Skoda
Kocsi ajtok szama: 5
Kocsi db szama    : 1
Kocsi ara         : 2500000

Kocsi tipusa      : Opel
Kocsi ajtok szama: 5
Kocsi db szama    : 2
Kocsi ara         : 2700000

Autokereskedes

Kocsik teljes ara      : 13900000.000000
Kocsik ossz darabszama : 5
```

14. Gyakorlat

Rövid elméleti összefoglaló

- **Beágyazás**: Egy osztály egy másik osztálynak lehet adattagja, amely azt jelenti, hogy a beágyazott osztály adattagjait, tagfüggvényeit használhatjuk. A beágyazott osztály adattagjainak **public** hozzáféréssel ajánlott rendelkezniük.

A beágyazott osztály adattagjainak és tagfüggvényeinek használata. (*MINTA14_01*)

```
//OSZTALY.H
class Szakasz
{
  public:
    int x1,y1,x2,y2; // két koordinátapont adatai (egyenes szakasz)
  public:
   // Paraméteres konstruktor
   Szakasz(int x11,int y11, int x21, int y21)
   {
    x1 = x11; y1 = y11; x2 = x21; y2 = y21;
   }
   Szakasz(const Szakasz& p) // Másoló konstruktor
   {
     x1 = p.x1; y1 = p.y1; x2 = p.x2; y2 = p.y2;
   }
   int GetX1(){ return x1;}
   int GetY1(){ return y1;}
   int GetX2(){ return x2;}
   int GetY2(){ return y2;}
};
class Tavolsag
{
  protected:
    double tav; // a  tav adattag a két pont távolságát tartalmazza
    double r;   // két ponton átmenő kör sugara
  public:
    Szakasz sz; // sz a Szakasz objektumpéldánya.
    Tavolsag (int a1, int b1, int a2, int b2)
    : sz(a1,b1,a2,b2) {tav = 0; r = 0;} // sz inicializálása
    Tavolsag (const Szakasz& p) : sz(p) // másoló konstruktor
    { tav = 0; r = 0; }
    void TavSzamol();
    void  SugarSzamol();
    double GetTav(){ return tav;}
    double GetSugar() { return r; }
};
```

```
//OSZTALY.CPP
#include <cmath>
#include "Osztaly.h"
// TavSzamol() tagfüggvény két koordinátapont távolságát számítja.
void Tavolsag::TavSzamol()
{
  tav = sqrt(pow(sz.x1-sz.x2,2) + pow(sz.y1-sz.y2,2));
}
```

```
// a két koordinátaponton átmenő kör sugarának számítása
void Tavolsag::SugarSzamol()
{
  r = tav/2;
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{
  Tavolsag t(2,2,4,4); // A t objektumpéldány inicializálása.
  t.TavSzamol();
  t.SugarSzamol();
  cout << "tavolsag : " << t.GetTav() << endl;
  cout << "sugar : " << t.GetSugar() << endl << endl;
  cout << "masolo konstruktor hasznalata " << endl;

  // A t1 objektumpéldány másoló konstruktorral való inicializálása.
  Tavolsag t1(t);
  t1.TavSzamol();
  t1.SugarSzamol();
  cout << "tavolsag : " << t1.GetTav() << endl;
  cout << "sugar : " << t1.GetSugar() << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
tavolsag : 2.82843
sugar : 1.41421

masolo konstruktor hasznalata
tavolsag : 2.82843
sugar : 1.41421
```

Általánositott osztályok (template):

- A paraméterezett (általánosított) osztály (**generic class**, vagy **class generator**) lehetővé teszi, hogy más osztályok definiálásához a paraméterezett osztályt, mint mintát használjuk. Ezáltal egy adott osztálydefiníció minden típus esetén alkalmazható.

- A sablonosztály általános alakja, ahol a *tipus1*, ...*tipusn* az általánosított típusokat jelölik:

```
template <class tipus1, … class tipusn > class osztalynev
{ };
```

- A osztály nem inline tagfüggvényeinek definíciója az alábbiak szerint kell megadnunk:

```
template <class tipus1, … class tipusn >
fvtípus osztalynev <tipus1,… tipusn> :: fvnev (parameterlista)
{ };
```

Sablon létrehozása a *Muvelet* osztály használatára. (*MINTA14_02*)

```
//MUV.H
template <class R>
class Muvelet
{
  private:
    R a,b,ered;
  public:
    Muvelet(R a1, R b1);
    void Szamol();
    R Eredmeny();
    void Kiir();
};
```

A paraméterek típusa lehet:

- **int**
- **float**
- **double**

A *Muvelet* osztály adattagjai:
a, b az operandusok,
ered az eredményt tartalmazza.

```
template <class R>
Muvelet<R>::Muvelet(R a1, R b1)
{
  a = a1; b = b1;
}
```

A konstruktor értéket ad az *a* és *b* adattagoknak.

```
template <class R>
void Muvelet<R>::Szamol()
{
  ered = (a+b);
}
```

A *Szamol()* tagfüggvény az *ered* adattagba kiszámítja az *a* és a *b* adattagok összegét.

```
template <class R>
R Muvelet<R>::Eredmeny()
{ return ered; }
```

Az *Eredmeny()* tagfüggvény az *ered* adattagot adja vissza.

```
template <class R>
void Muvelet<R>::Kiir()
{ cout << "eredmeny: " << ered
       << endl;
}
```

A *Kiir()* tagfüggvény szövegesen írja ki az eredményt.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "muv.h"
int main()
{
  Muvelet<double> w1(1.5,3.5);
  cout.setf(iostream::fixed,
      iostream::floatfield);
  w1.Szamol();
  cout << "ered tartalma: "
       << w1.Eredmeny()<< endl;
  w1.Kiir();
  Muvelet<int> w2(1,2);
  w2.Szamol();
  cout << "ered tartalma: "
       << w2.Eredmeny()
       << endl;
  w2.Kiir();
  cin.get();
  return 0;
}
```

A *w1* objektumpéldány inicializálása a konstruktor **double** típusú argumentumokkal való hívásával.

A *w2* objektumpéldány inicializálása a konstruktor **int** típusú argumentumokkal való hívásával.

A program futásának eredménye:

```
ered tartalma: 5.000000
eredmeny: 5.000000

ered tartalma: 3
eredmeny: 3
```

- Saját névterület kialakítása és használata

  namespace *névtér* {
  deklarációk és definíciók
  }

  A közös névterületen definiálhatunk konstansokat, változókat, függvények prototípusát. Példában a ***main()*** függvény a közös **namespace** területről csak a *PI* konstanst, az *eredmeny* változót, és a *negyzet()* függvényt használja.

A közös névterület használata. (*MINTA14_03*)

```
// kozos.h
namespace kozos
{
  const double PI = 3.141582654;
  double eredmeny;
  int adat;
  double negyzet (const double x){ return x*x; }
  double kob (const double y){ return y*y*y; }
}
```

```
//MINTA14_03
#include <iostream>
using namespace std;
#include <stdlib.h>

#include "kozos.h"
using  kozos::PI;
using  kozos::negyzet;
using  kozos::eredmeny;

int main()
{
  double r = 1, x = 3;
  eredmeny = negyzet(x);
  cout << "negyzete: " << eredmeny << endl;
  eredmeny = r *r *PI;
  cout << "kor terulete: " << eredmeny << endl;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
negyzete: 9
kor terulete: 3.14158
```

Példa a teljes **namespace** használatára (*MINTA14_04*).

```
// SAJAT.H
namespace Sajat
{ // függvények prototípusa
  void KeresMaxMin (int *a,int n,int& max,int& min);
  void Lista (const int *a, int n);
}
```

```
//SAJAT.CPP
#include <iostream>
using namespace std;
```

```
#include "Sajat.h" // saját függvények deklarálása

using namespace Sajat;
// névtelen névtér modulszintű definíciókhoz
namespace {
  const char TAB='\t';
  const double PI = 3.141592654;
}

// Az a tömb elemeiből kikeressük a max és a min értékeket.
void Sajat::KeresMaxMin (int *a, int n, int& max, int& min)
{
  max = a[0];
  min = a[0];
  for ( int i = 1; i<n; i++)
  {
    if (a[i]> max) max = a[i];
    if (a[i] < min) min = a[i];
  }
}

// Az a tömb elemeinek megjelenítése.
void Sajat::Lista (const int *a, int n)
{
  for (int i = 0; i<n; i++)
     cout << a[i] << TAB;
  cout << endl;
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include <stdlib.h>

#include "Sajat.h"
using namespace Sajat;
// A teljes Sajat namespace használata.

int main()
{
  int a[5], min, max;
  cout << "5 db egesz adat megadasa: \n";
  for (int i = 0; i<5; i++)
  { // tömb feltöltése
    cout << "adat: "; cin >> a[i];
  }
  KeresMaxMin(a,5,max,min);
  Lista(a,5);
  cout << "min adat: " << min << endl;
  cout << "max adat: " << max << endl;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
5 db egesz adat megadasa:
adat: 9
adat: 2
adat: -1
adat: 4
adat: 0
9    2    -1    4    0
min adat: -1
max adat: 9
```

Feladatok

1. Használjuk az alábbi definíciókat!

```
//OSZTALY.H
class diak
{
 public:
   char nev[15];
   int tankor;
   int kredit;
   double atlag;
 public:
   diak();
};
```

```
class evfolyam
{
  protected:
    int letszam;
    diak* evf;
    double EvfAtlag;
    int MaxKredit;
    int MinKredit;
    double MaxAtlag;
  public:
    evfolyam();
    ~evfolyam();
    void Ertekel();
    double GetMaxAtlag() { return EvfAtlag; }
    int GetMaxKredit() { return MaxKredit; }
    int GetMinKredit() { return MinKredit; }
};
```

```
int main()
{
  evfolyam *evf1;
  evf1 = new evfolyam;
  evf1->Ertekel();
    . . .
}
```

Az *Ertekel()* tagfüggvény számítsa ki az évfolyam átlagát, keresse meg a legnagyobb, ill. a legkisebb kreditpontot! Dolgozzuk ki a tagfüggvényeket az *OSZTALY.CPP* állományban, és a ***main***() függvényben jelenítsük meg az eredményeket. (GYAK14_1)

Segítség a konstruktor tervezéshez:

```
evfolyam::evfolyam()
{
    cout << "Diakok szama: "; cin >> letszam;
    evf = new diak[letszam];
    for (int i = 0; i< letszam; i++)
    {
      cout << endl;
      cout << "Neve   : "; cin >> evf[i].nev;
      cout << "Tankor: "; cin >> evf[i].tankor;
      cout << "Kredit: "; cin >> evf[i].kredit;
      cout << "Atlag : "; cin >> evf[i].atlag;
    }
}
```

Destruktor tervezése:

```
evfolyam::~evfolyam()
{
  delete [] evf;
}
```

2. Használjuk az alábbi definíciókat!

```
//OSZTALY.H
class Adatok
{
  public:
    double a, b, atlag;
  public:
  Adatok(double a1, double b1)
  {
    a = a1; b = b1; atlag = 0;
  }
  void AtlagSzamol();
  double GetAtlag()
     { return atlag; }
  double Geta() { return b; }
  double Getb() { return a; }
};
```

```
class Vegrehajt
{
  protected:
    double m_kozep;
  public:
   Adatok t;
   Vegrehajt(double a1, double b1)
             :t(a1,b1)
   { m_kozep = 0; }
   void MertaniKozep();
   double GetMk(){ return m_kozep;}
   double GetSzk()
   { t.AtlagSzamol();
      return t.GetAtlag();
   }
};
```

Dolgozzuk ki a tagfüggvényeket az *OSZTALY.CPP* állományban, és írjuk meg a ***main***() függvényt! (GYAK14_2)

3. Használjuk az alábbi állományokat a *Sajat* **namespace** kialakításához! (GYAK14_3)

```
// SAJAT.H
namespace Sajat
{
 void KeresParosPt(int *x, int db, int& par, int& pt);
 void TombKiir(const int *x, int db);
}
//SAJAT.CPP
#include <iostream>
#include <iomanip>
using namespace std;
#include "Sajat.h" // saját függvények prototípusa
using namespace Sajat;
void Sajat::KeresParosPt(int *x, int db, int& par, int& pt)
{
  par = pt = 0;
  for (int i = 0; i<db; i++)
  {
    if (x[i] % 2 != 0) pt++;
    else par++;
  }
}
void Sajat::TombKiir(const int *x, int db)
{
  for (int i = 0; i<db; i++)
    cout<< setw(5) << x[i];
  cout << endl;
}
```

Olvassuk be az a tömb elemeit, aktiváljuk az alábbi két függvényt:

```
KeresParosPt(w,n,ps,pt);
TombKiir(w,n);
```

és jelenítsük meg a *ps* és a *pt* értékeit!

```
//GYAK14_3
#include <iostream>
using namespace std;
#include <stdlib.h>
#include "Sajat.h"
using namespace Sajat;
int main()
{ int w[20], ps, pt, n;
    . . .
}
```

4. Használjuk az alábbi definíciót egy olyan osztálysablon tervezésére, amely alkalmas arra, hogy a ***main***() függvényben **float**, **double**, illetve **int** típusú adattagokkal rendelkező objektumpéldányokat hozzunk létre! (GYAK14_4)

```
//OSZTALY.H
template <class R>
class Teglatest
{ private:
    R a,b,c,felsz,terf;
  public:
    Teglatest(R a1, R b1, R c1);
    void Terfogat();
    void Felszin();
    R GetFelszin();
    R GetTerfogat();
};
```

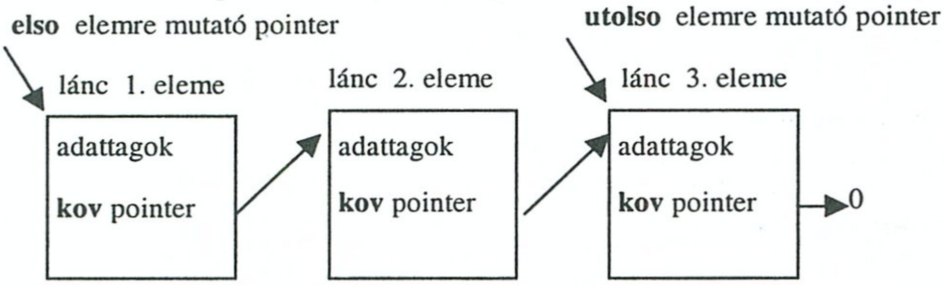
Dolgozzuk ki a tagfüggvényeket az *OSZTALY.H* állományban! A ***main()*** függvényben aktiváljuk valós és egész típusú argumentumokkal az objektumpéldány konstruktorát és tagfüggvényeit, jelenítsük meg az eredményeket!
Segítség a ***main()*** függvény kidolgozásához:

```
int main()
{ cout << "Valos tipusu parameterek" << endl;
  Teglatest<double> t1(1.0,2.0,3.0);  // konstruktor hivasa
  cout.setf(iostream::fixed, iostream::floatfield);
  t1.Terfogat(); t1.Felszin();
  cout << "Teglatest terfogata: " << t1.GetTerfogat() << endl;
  cout << "Teglatest felszine : " << t1.GetFelszin() << endl;
  cout << endl << "Egesz tipusu parameterek" << endl;
  Teglatest<int> t2(1,2,3);  // konstruktor hivasa
  t2.Terfogat(); t2.Felszin();
  cout << "Teglatest terfogata: " << t2.GetTerfogat() << endl;
  cout << "Teglatest felszine : " << t2.GetFelszin() << endl;
  cin.get();
  return 0;
}
```

15. Gyakorlat

Rövid elméleti összefoglaló

- Egyirányú láncolt lista. Az láncelem csak a következő elemre mutató pointert tárolja (*kov*). A mintafeladatban a *kov* pointer *Teglalap* típusú osztályra tud mutatni. (*MINTA15_01*)


elso elemre mutató pointer
utolso elemre mutató pointer
lánc 1. eleme
lánc 2. eleme
lánc 3. eleme
adattagok
kov pointer
adattagok
kov pointer
adattagok
kov pointer
0

Az OSZTALY.H állományban lévő osztályok ismertetése:

A *Teglalap* osztály adattagjai:

- *a,b* a téglalap két oldala,
- *ter* a téglalap kerülete,
- *ter* a téglalap területe.

Tagfüggvényei:
- paraméteres konstruktor,
- destruktor,
- *Szamitas()*, amely a téglalap kerületét és területét számítja ki.

Lista osztály adattagjai:
- *elso* a láncolt lista első elemére mutat
- *utolso* a láncolt lista utolsó elemére mutat.

Az elemek a *Teglalap* osztalyra mutató pointerek.

Tagfüggvényei:
- konstruktor,
- destruktor,
- *HozzaAd()*, melynek paraméterei a téglalap *a, b* oldala és egy egész típusú *irany*, amely meghatározza a hozzáfűzés irányát: 1 akkor a lista elejére fűzi, ha nem 1, akkor a végére.
- *MaxTer()* megkeresi a maximális területű téglalapot,
- *MinKer()* megkeresi a minimális kerületű téglalapot,
- *Kiir()* kiírja a láncolt lista tartalmát.

```
//OSZTALY.H
class Teglalap
{
  public:
    double a,b,ter,ker;
    Teglalap *kov;
  public:
    Teglalap(double a1, double b1);
    ~Teglalap();
    void Szamitas();
};
```

```
class Lista
{
  private:
    Teglalap *elso;
    Teglalap *utolso;
  public:
    Lista();
    ~Lista();
    void HozzaAd(double a1, double b1, int irany);
    void MaxTer();
    void MinKer();
    void Kiir();
};
```

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
Teglalap::Teglalap(double a1, double b1)
{
  a = a1; b = b1; // téglalap oldalainak inicializálása
}
Teglalap::~Teglalap(){ }
void Teglalap::Szamitas()
{
  ter = a*b;  ker = 2*(a+b); // terület és kerület számítása
}
Lista::Lista()
{
  elso = 0; utolso = 0;
}
Lista::~Lista()
{
  Teglalap *t, *t1;
   t = elso;
   while ( t != 0)  // felszabadítjuk a lista elemeit
   {                // míg a lánc végére nem érünk
     t1 = t->kov;
     delete t; t = t1;
   }
   elso = 0; utolso = 0;
}
// új adattal bővítjük a láncot
void Lista::HozzaAd(double a1, double b1, int irany)
{
  Teglalap *t;
  if (elso == 0)
  {
    elso = new Teglalap(a1,b1); // most létrehozott elem lesz
    elso->Szamitas();           // a lánc első eleme
    utolso = elso;
  }
  else if (irany == 1)   // a lánc elejére fűz
  {
    t = new Teglalap(a1, b1);
    t->Szamitas();
    t->kov = elso; elso = t;
  }
  else   // a lánc végére fűz
  {
    t = new Teglalap(a1, b1);
    t->Szamitas();
    utolso->kov = t;
    utolso = t;
  }
}
void Lista::MaxTer() // legnagyobb terület keresése
{                    // a hozzá tartozó oldalakkal
  Teglalap *t;
  double a1, b1, ter1;
  cout <<endl <<"Maximalis terulet"<< endl;
  t = elso;
  ter1 = 0;
  while (t != 0)
  {
    if (t->ter > ter1)
    {
      a1 = t->a; b1 = t->b; ter1=t->ter;
    }
    t = t->kov;
  } // a megtalált adatok kiíratása
  cout << "a: " << a1 << " b: " << b1 << " terulet: "
       << ter1 << endl;
}
void Lista::MinKer() // legkisebb kerület keresése
{                    // a hozzá tartozó oldalakkal
  Teglalap *t;
  double a1, b1, ker1;
  cout << endl << "Minimalis kerulet "
       << endl;
  t = elso;
  if (t != 0) ker1 = t->ker;
  while (t != 0)
  {
    if (t->ker < ker1)
    {
      a1 = t->a; b1 = t->b; ker1=t->ter;
    }
    t = t->kov;
  }  // a megtalált adatok kiíratása
  cout << "a: " << a1 << " b: " << b1
       << " kerulet: " << ker1 << endl;
}
```

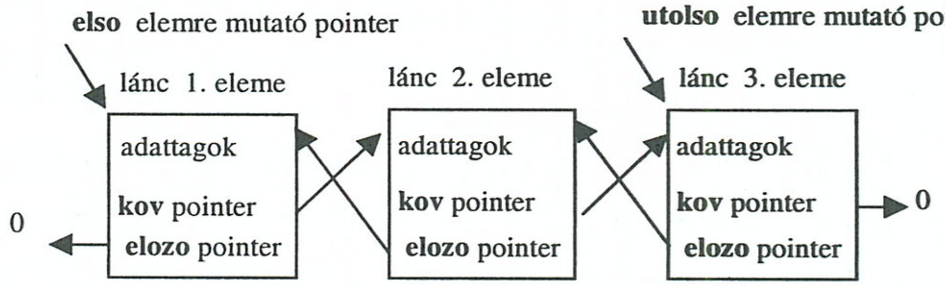
```
void Lista::Kiir()
{
  Teglalap *t;
  cout <<endl<<"Teglalapok listaja"<< endl;
  t = elso;
  while (t != 0)
  { // a téglalap oldalainak, kerületének és területének kiírása
    cout << "a: " << t->a << " b: " << t->b
         << "  kerulet: " << t->ker
         << "  terulet: " << t->ter <<endl;
    t = t->kov;
  }
}
```

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{
  Lista *tg;
  tg = new Lista();
  tg->HozzaAd(6,2,1);
  tg->HozzaAd(3,4,1);
  tg->HozzaAd(5,2,0);
  tg->HozzaAd(3,7,1);
  tg->HozzaAd(8,3,1);
  tg->Kiir();
  tg->MaxTer();
  tg->MinKer();
  delete tg;
  cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Teglalapok listaja
a: 8 b: 3  kerulet: 22  terulet: 24
a: 3 b: 7  kerulet: 20  terulet: 21
a: 3 b: 4  kerulet: 14  terulet: 12
a: 6 b: 2  kerulet: 16  terulet: 12
a: 5 b: 2  kerulet: 14  terulet: 10

Maximalis terulet
a: 8 b: 3 terulet: 24

Minimalis kerulet
a: 3 b: 4 kerulet: 12
```

- **Kétirányú láncolt lista.** A lánc elemei nemcsak a következő elemet ismerik (*kov*), hanem az előzőt (*elozo*) is. Listaelem törlésének bemutatása. (*MINTA15_02*)


elso elemre mutató pointer
utolso elemre mutató pointer
lánc 1. eleme
lánc 2. eleme
lánc 3. eleme
adattagok
kov pointer
elozo pointer
adattagok
kov pointer
elozo pointer
adattagok
kov pointer
elozo pointer
0
0

```
//OSZTALY.H
class Teglalap
{ public:
    double a,b,ter,ker;
    Teglalap *kov;
    Teglalap *elozo;
```

A *Teglalap* osztály tartalmaz egy *elozo* adattagot, amely az a lista előző elemét kezeli.

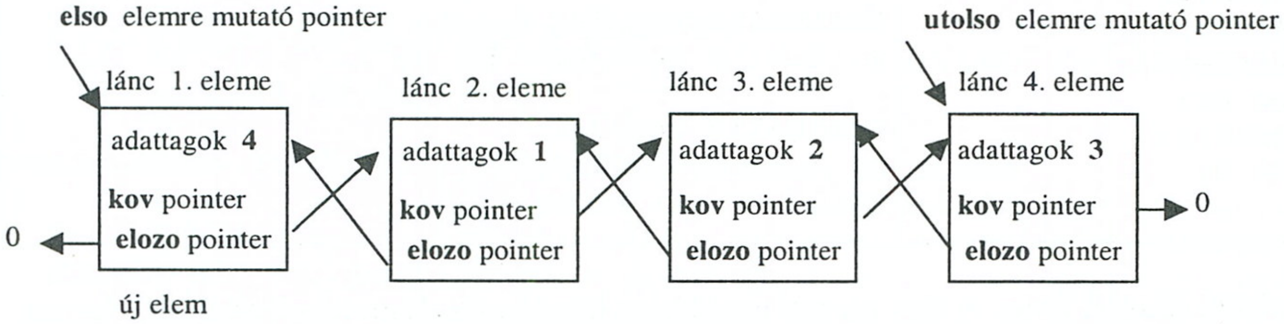
```
  public:
    Teglalap(double a1, double b1);
    ~Teglalap();
    void Szamitas();
};
class Lista
{
  private:
    Teglalap *elso;
    Teglalap *utolso;
  public:
    Lista();
    ~Lista();
    void HozzaAd(double a1, double b1,
                 int irany);
    void Torles(double a1, double b1);
    void MaxTer();
    void MinKer();
    void KiirElore();
    void KiirHatra();
};
```

Lista osztály bővült egy *Torles()* tagfüggvénnyel, mellyel a lista elemeit egyenként törölhetjük.

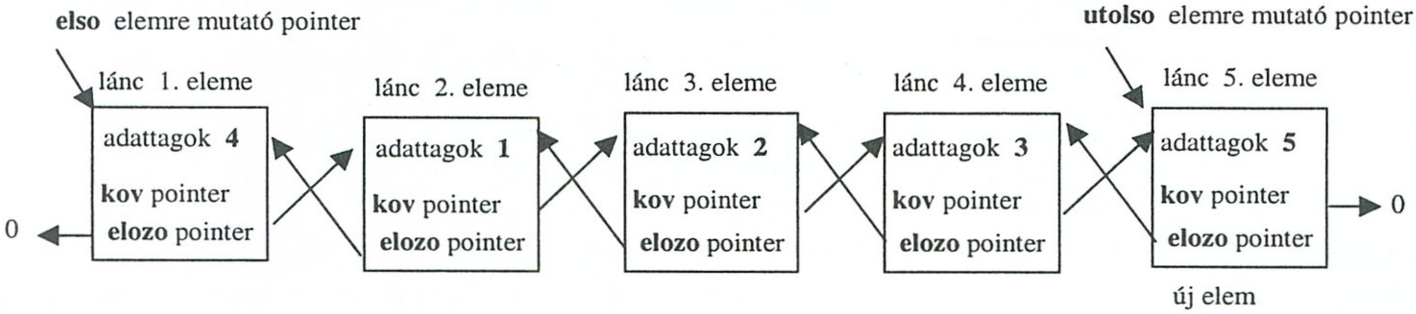
Lehetőség van a lista elemeit kiíratni előre a *KiirElore()* és visszafelé a *KiirHatra()* tagfüggvényekkel.

Természetesen vizsgálatot kell végezni, hogy a törlendő elem a lánc elején vagy a végén helyezkedik el, vagy a lánc közbülső eleme.

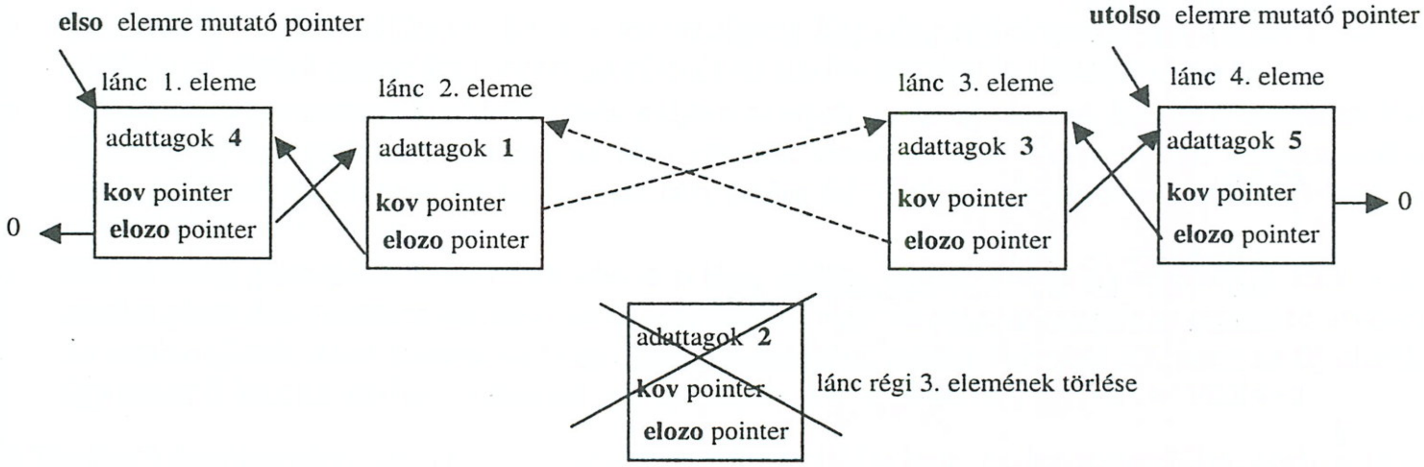
Hozzáadás a lánc elejére:


elso elemre mutató pointer
utolso elemre mutató pointer
lánc 1. eleme
lánc 2. eleme
lánc 3. eleme
lánc 4. eleme
adattagok **4**
kov pointer
elozo pointer
adattagok **1**
kov pointer
elozo pointer
adattagok **2**
kov pointer
elozo pointer
adattagok **3**
kov pointer
elozo pointer
0
0
új elem

Hozzáadás a lánc végére:


elso elemre mutató pointer
utolso elemre mutató pointer
lánc 1. eleme
lánc 2. eleme
lánc 3. eleme
lánc 4. eleme
lánc 5. eleme
adattagok **4**
kov pointer
elozo pointer
adattagok **1**
kov pointer
elozo pointer
adattagok **2**
kov pointer
elozo pointer
adattagok **3**
kov pointer
elozo pointer
adattagok **5**
kov pointer
elozo pointer
0
0
új elem

Törlés a lánc közepéből:


elso elemre mutató pointer
utolso elemre mutató pointer
lánc 1. eleme
lánc 2. eleme
lánc 3. eleme
lánc 4. eleme
adattagok 4
kov pointer
elozo pointer
adattagok 1
kov pointer
elozo pointer
adattagok 3
kov pointer
elozo pointer
adattagok 5
kov pointer
elozo pointer
0
0
adattagok 2
kov pointer
elozo pointer
lánc régi 3. elemének törlése

```
//OSZTALY.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
Teglalap::Teglalap(double a1, double b1)
{ a = a1; b = b1; kov = elozo = 0; }
Lista::Lista()
{ elso = 0; utolso = 0; }
void Lista::HozzaAd(double a1, double b1, int irany)
{
  Teglalap *t;
  if (elso == 0)
  {
    elso = new Teglalap(a1,b1); elso->Szamitas();
    utolso = elso;
  }
  else if (irany == 1)
  {
     t = new Teglalap(a1, b1); t->Szamitas();
     t->kov = elso; elso->elozo = t; elso = t;
  }
  else
  {
    t = new Teglalap(a1, b1); t->Szamitas();
    utolso->kov = t; t->elozo = utolso; utolso = t;
  }
}
void Lista::Torles(double a1, double b1)
{
  Teglalap *t, *t1, *t2;
  t = elso;
  while ( t != 0)
  {
    if (t->a == a1 && t->b == b1)
    {
      t1 = t->elozo; t2 = t->kov;
      if (t1  == 0)
      {
        elso = t2;
        if( t2 != 0) t2->elozo = 0;
        else utolso = t2;
      }

      else
      {
        t1->kov = t2;
        if (t2 != 0) t2->elozo = t1;
        else utolso = t1;
      }
      t->kov = 0; t->elozo = 0; delete t; break;
    }
    t = t->kov;
  }
}
void Lista::KiirElore()
{
  Teglalap *t;
  cout <<endl
  <<"Teglalapok listaja elore " << endl;
  t = elso;

  while (t != 0)
  { cout << "a: " << t->a << " b: " << t->b
         << "  kerulet: " << t->ker
         << "  terulet: " << t->ter << endl;
    t = t->kov;
  }
}
void Lista::KiirHatra()
{
  Teglalap *t;
  cout << endl <<"Teglalapok listaja visszafele" << endl;
  t = utolso;

  while (t != 0)
  { cout << "a: " << t->a << " b: " << t->b
         << "  kerulet: " << t->ker
         << "  terulet: " << t->ter << endl;
    t = t->elozo;
  }
}
```

Megjegyzések a tagfüggvényekhez:
A *Teglalap()* konstruktor értéket ad az *a* és a *b* adattagoknak az argumentumlistáról és nullázza a *kov* és az *elozo* mutatókat.

A *Lista()* konstruktor alapállapotba állítja az *elso* és az *utolso* mutatókat.

A *HozzaAd()* tagfüggvénynek az adatok beszúrásakor kezelni kell az *elozo* mutatót is.

- Ha az *elso* pointer értéke 0, akkor a megszületett *Teglalap* objektumpéldány a lánc első és egyben az utolsó eleme lesz, ezért az *elso* és az *utolso* pointer is rámutat.
- Ha az *irany* paraméter 1, akkor a lánc elejére szúrjuk be az új elemet. Ekkor a megszületett új elem (*t*) *kov* pointere rámutat az *elso* elemre, majd az *elso elozo* eleme visszamutat a most született (*t*) elemre, és az *elso* pointer pedig felveszi a *t* címét. Azaz most a *t* lesz az első.
- Ha az *irany* paraméter 0 (nem 1), akkor a lánc végére fűzzük be az új elemet (*t*). Ekkor az *utolso* elem *kov* pointere rámutat az új (*t*) elemre, viszont a *t* új elem *elozo* pointere átveszi az *utolso* címét, azaz visszamutat az utolsóra. Majd az *utolso* felveszi a *t*, azaz az új címét, így most ő lesz az utolsó.

A *Torles()* tagfüggvény az *a1* és *b1* adatot megkeresi láncban, és ha megtalálta, törli a lánc elemét. Természetesen figyelni kell arra, hogy a törlendő elem a lánc elején, a lánc belsejében vagy a végén van. A lánc végét a 0 pointer jelzi.

- A *t* pointerbe felvesszük a lánc első elemére mutató pointert (*elso*). A **while** ciklus feltétele ellenőrzi, hogy van-e még eleme a láncnak, ekkor a *t* értéke nem 0. A lánc végét a 0 jelzi.
- A megtalált elemnél *t1* változóba tároljuk a *t* elem *elozo*, a *t2* változóba a *t* elem *kov* pointerét.
- Ha a *t1*, azaz a *t elozo* pointere 0, akkor nincs előző eleme, tehát ő az első elem, ezért az *elso* átveszi a *t2*, azaz *t* következő láncra mutató címét. Ha a *t2* nem nulla, az *elozo* pointerét nullázzuk.
  Ha a *t2* 0, akkor a *t2* lesz az *utolso* láncelem, azaz a *t* törlése után üres lesz a lánc.
- Ha nem első elem, akkor a *t elozo* elemének a *kov* mezejébe a *t* következő elemének a címét írjuk bele. Ha a *t2* nem nulla, ez azt jelenti, hogy a lánc belsejében vagyunk, ezért a *t* következő elemének az *elozo* elemébe írjuk a *t elozo* elemre mutató címét (*t1*).
  Ha a *t2* 0, az azt jelenti, hogy a törlendő elem az utolsó elem, ezért a *t1* lesz az *utolso* láncelem.
- Nullázzuk a *t elozo* és *kov* tagját, és a **delete** operátorral töröljük a *t* elemet.

A *KiirElore()* tagfüggvény az *elso* elemtől indul el, listázza az adatokat, addig veszi a következőt, míg a lánc végére nem értünk.

A *KiirHatra()* tagfüggény az *utolso* pointertől indul el, és hátulról előre listázza az elemeket. Az *elozo* pointereken lépked visszafelé, míg a 0 nem lesz az értéke.

A *MINTA15_02* mintafeladatban a főfüggvény bemutatja a *HozzaAd()* tagfüggvény aktiválását a lánc elejére és végére való fűzéssel. A *KiirasElore()* tagfüggvénnyel ellenőrizzük a lánc tartalmát. A *Torles()* tagfüggvénnyel törlünk a láncból.

```
//MAIN.CPP
#include <iostream>
using namespace std;
#include "Osztaly.h"
int main()
{ Lista *tg;
  tg = new Lista();
  tg->HozzaAd(6,2,1);
  tg->HozzaAd(5,2,0);
```

A program futásának eredménye:

```
Teglalapok listaja elore
a: 3 b: 7  kerulet: 20  terulet: 21
a: 6 b: 2  kerulet: 16  terulet: 12
a: 5 b: 2  kerulet: 14  terulet: 10

Torles : 5, 2
```

```
  tg->HozzaAd(3,7,1);
  tg->KiirElore();
  cout << "\nTorles : 5, 2 ";
  tg->Torles(5,2);
  tg->KiirElore();
  delete tg;
  cin.get();
  return 0;
}
```

```
Teglalapok listaja elore
a: 3 b: 7  kerulet: 20  terulet: 21
a: 6 b: 2  kerulet: 16  terulet: 12
```

A *MINTA15_03* tartalmazza a *MINTA15_02* feladat sablonos megoldását.

Feladatok

1. Tervezzük osztályt (*Adatok*) két valós adat számtani közepének kiszámítására, és egy *Lista* osztályt, amely az objektumpéldányokat csak elore mutató (*kov*) pointerrel láncba fűzi. (GYAK15_1)

```
//OSZTALY.H
class Adatok
{
  public:
    double x,y, szk;
    Adatok *kov;
  public:
    Adatok(double x1,
           double y1);
    ~Adatok();
    void SzamtaniKozep();
};
```

```
class Lista
{
  private:
    Adatok *elso;
    Adatok *utolso;
  public:
    Lista();
    ~Lista();
    void HozzaAd(double x1,double y1);
    void AtlagokAtlaga();
    void SzkLista();
};
```

Az *Adatok* osztály *SzamtaniKozep()* tagfüggvényében számítsuk ki a két valós adattag számtani közepét!

A *Lista* osztály *HozzaAd()* tagfüggvényében a paraméterlistán átvett két valós adattal hozzunk létre egy *Adat* objektumpéldányt, és aktiválva a *Szamtanikozep()* tagfüggvényt, számítsuk ki a számtani közepüket, majd fűzzük egyirányú láncba az objektumpéldányt!

Az *AtlagokAtlaga()* tagfüggvényben számítsuk ki a lánc elemei átlagának az átlagát!

Az *SzkLista()* tagfüggvénnyel jelenítsük meg a lánc elemeinek adattagjait!

A program futásának eredménye:

```
Adatok listaja
         x         y szamtani kozep
-----------------------------------
         1         2      1.5
         4         6        5
         7         9        8
        10        12       11

Atlagok atlaga: 6.375
```

2. Tervezzünk derékszögű háromszög adatainak tárolására egy osztályt, valamint egy *Lista* osztályt, amely láncba fűzi a derékszögű háromszögeket! Használjuk az alábbi definíciókat! (GYAK15_2)

```
class DSzH // derékszögű háromszög
{ public:
    double a,b,c,ter,ker;
    DSzH *kov;
    DSzH *elozo;
  public:
    DSzH(double a1, double b1);
    ~DSzH();
    void Szamol();
};
class Lista
{ private:
    DSzH *elso;
    DSzH *utolso;
  public:
    Lista();
    ~Lista();
    void HozzaAd(double a1, double b1);
    void MaxAtfogo();
    void TerOsszeg();
    void ListaElore();
};
```

A *DSzH* osztály *Szamol()* tagfüggvényében számítsuk ki a derékszögű háromszög átfogóját, a kerületét és a területét!

A *Lista* osztály *HozzaAd()* tagfüggvényében az argumentumlistán megadott befogókkal hozzuk létre a *DSzH* objektumpéldányt, aktiváljuk a *Szamol()* tagfüggvényt, majd az új elemet fűzzük a lánc végére!

A *MaxAtfogo()* tagfüggvényben keressük meg a legnagyobb átfogójú derékszögű háromszöget, de a két befogót is írassuk ki!

A *TerOsszeg()* tagfüggvényben adjuk össze a derékszögű háromszögek területeit és jelenítsük meg.

A *ListaElore()* tagfüggvényben listázzuk a lánc elemeinek adatait táblázatos formában!

A ***main()*** függvény részlete:

```
Lista *th;
th = new Lista();
th->HozzaAd(3,4);
th->HozzaAd(1,3);
th->HozzaAd(2,4);
th->ListaElore();
th->TerOsszeg();
th->MaxAtfogo();
```

A program futásának eredménye:

```
Derekszogu haromszogek adatai
         a         b         c   terulet
----------------------------------------
         3         4         5        12
         1         3   3.16228         3
         2         4   4.47214         8

Teruletek osszege: 23

Maximalis atfogo
a oldal: 3 b oldal: 4 atfogo: 5
```

16. Gyakorlat

Rövid elméleti összefoglaló

A C++ nyelvben a kivételkezelés lehetőségei:

- A kivételkezelés programrészletének kijelölése (**try**).
- A kivételek továbbítása (**throw**).
- A kivételek elfogása és kezelése (**catch**).

A mintafeladatokban a tömbök memóriafoglalásakor előforduló memóriahiányt kétféleképpen figyelhetjük. A C++ nyelv kevés eleme használja a kivételkezelést, így általában magunknak kell gondoskodni a kivételek előállításáról és a kezeléséről egyaránt.

```
//MINTA16_01
#include <iostream>
#include <new>   // a nothrow argumentum használatához szükséges
using namespace std;

int main()
{
  int *x;
  x = new(nothrow) int[1000000000];
  if (!x)
  {
    cerr << "Nincs eleg memoria!" << endl;
    cin.get(); cin.get();
    exit(-1); // Kilépés a programból.
  }
  // A lefoglalt memória törlése
  delete []x;
  cin.get(); cin.get();
  return 0;
}
```

A program futásának eredménye:

```
Nincs eleg memoria!
```

Megjegyzés: Ha a **new** operátor után megadjuk a *nothrow* argumentumot, akkor az operátor a kivétel létrehozása helyett 0 értékkel tér vissza, amit megvizsgálunk.

```
//MINTA16_02
#include <iostream>
#include <new>
using namespace std;
int main()
{
  int *x;
  try //A try megpróbálja végrehajtani.
  { x = new int[100000000]; }
  catch (bad_alloc)
//A catch elfogja és a paraméternek megfelelően kezeli a kivételt.
  {
  // sikertelen foglalás
   cerr << "Nincs eleg memoria!" << endl;
   cin.get();
   exit(-1); // Kilépés a programból.
  }
  // A lefoglalt memória törlése
  delete []x;
  cin.get(); cin.get();
  return 0;
}
```

A program futási eredménye:

```
Nincs eleg memoria!
```

A C++ nyelvben tetszőleges típushoz rendelhetünk kivételes helyzetet. Az alábbi példában a másodfokú egyenlet megoldása során a hibás helyzetek kezelését egyetlen helyen, a ***main()*** függvényben végezzük.

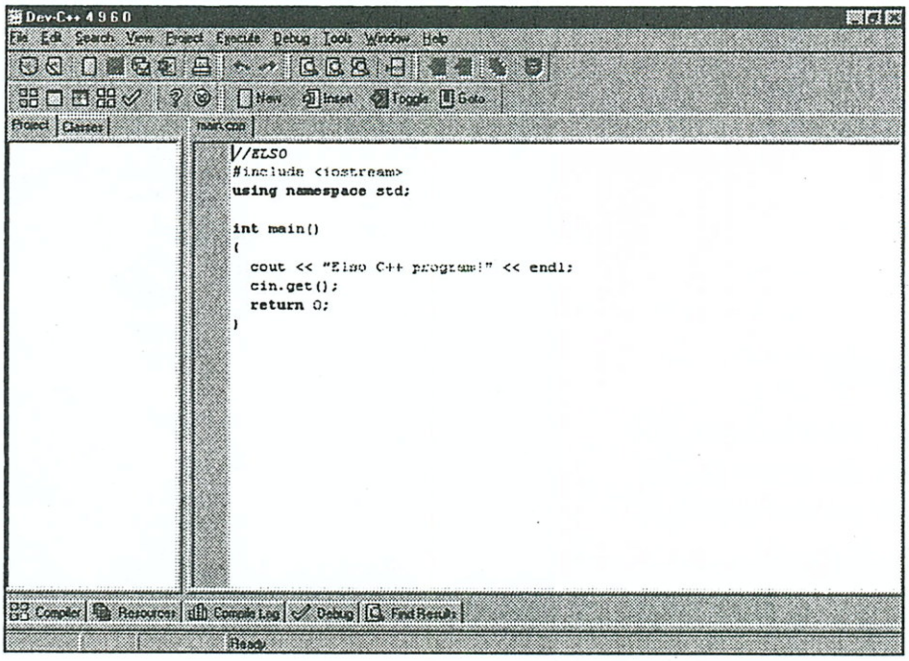
```
//MINTA16_03
#include <iostream>
#include <cmath>
using namespace std;
// Saját kivételtípusok készítése.
class EnemMasodfoku {};
class EnincsValosGyok {};
// A beolvasás során ENemMasodfoku * típussal jelezzük, ha a másodfokú
// tag együtthatója 0. Csak ezt a kivételt továbbítja a függvény.
void Beolvas(double& a, double& b, double& c)
  throw (ENemMasodfoku *)
// Az EnemMasodfoku * típusú kivétel létrehozása.
{
  cout<<" a negyzetes tag egyutthatoja : ";
  cin>>a;
  cout<<" az elsofoku tag egyutthatoja : ";
  cin>>b;
  cout<<" a szabad tag                 : ";
  cin>>c;
  if (!a)
     throw new ENemMasodfoku;
}
void Gyokok(double a, double b, double c)
   throw (ENincsValosGyok *)
// Az EnincsValosGyok * típussal jelezzük, ha nincs valós gyöke
// az egyenletnek. Csak ezt a kivételt továbbítja a függvény.
{
   double d=b*b-4*a*c, x1, x2;
   if (d<0)
      // Az EnincsValosGyok * típusú kivétel létrehozása.
      throw new ENincsValosGyok;
   else if (d>0)
   {
      x1 = (-b-sqrt(d))/(2*a);
      x2 = (-b+sqrt(d))/(2*a);
      cout<<x1<<endl<<x2<<endl;
   }
   else
   {
      x1 = -b/(2*a);
      cout<<x1<<endl;
   }
}

int main()
{
   double a,b,c;
   // Az egyenletmegoldás lépéseit kivételek figyelésével végezzük.
   try
   {
      // Felkészülünk a lehetséges kivételek "elkapására"
      cout<<" A masodfoku egyenlet egyutthatoi:" << endl;
      Beolvas(a,b,c);
      Gyokok(a,b,c);
   }
   catch(ENemMasodfoku *p)
   {
     // Az EnemMasodfoku * típusú kivétel elfogása.
     cerr<<"Az egyenlet nem masofoku!"<<endl;
   }
   catch(ENincsValosGyok *)
   {
     // Az EnincsValosGyok * típusú kivétel elfogása.
     cerr<<"Az egyenletnek nincs valos gyoke!"<<endl;
   }
   catch(...)
   {
     // Minden más kivétel elkapása.
     cout<<"Valami mas hiba lepett fel!"<<endl;
   }
   cin.get();
   cin.get();
   return 0;
}
```

A program futásának eredménye:

```
A masodfoku egyenlet egyutthatoi:
 a negyzetes tag egyutthatoja : 1
 az elsofoku tag egyutthatoja : 1
 a szabad tag                 : -12
-4
3
```

F3. A CD-melléklet tartalomjegyzéke

\Acrobat	Az ***Adobe Acrobat Reader*** telepítő-állományai: \Windows 2000, XP\AdbeRdr602_hun_full.exe, \Windows 98 SE\AdbeRdr60_hun.exe
\Algoritmusok	A könyv **8.** és **18.** fejezetében használt matematikai módszerek részletes leírása:
\Integral.pdf	Numerikus integrálás (numerikus kvadratúra)
\Interplacio.pdf	A függvények közelítési módszerei
\Masodfoku.pdf	Másodfokú egyenlet megoldása
\Nemlin.pdf	Egyismeretlenes nemlineáris egyenlet megoldása
\Rendezes.pdf	Rendezési módszerek
\DevCpp	A ***Dev-C++*** **Win98/2000/XP** alatt futtatható, szabadon terjeszthető C/C++ fordító: *devcpp4970-gcc295.exe*

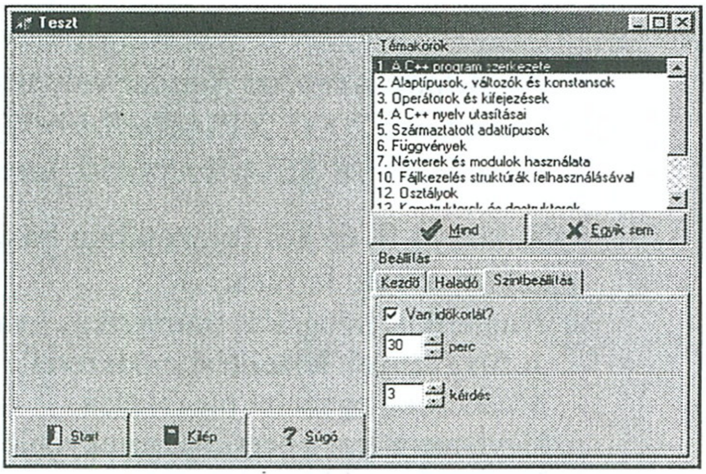

Dev-C++ 4.9.6.0
File Edit Search View Project Execute Debug Tools Window Help
New
Insert
Toggle
Goto
Project
Classes
main.cpp
Compiler
Resources
Compile Log
Debug
Find Results
Ready

```
//ELSO
#include <iostream>
using namespace std;

int main()
{
  cout << "Elso C++ program!" << endl;
  cin.get();
  return 0;
}
```

\Info	A CD-melléklet tartalmával kapcsolatos információk
\Programok	A könyvhöz tartozó forrásprogramok:
\Fejezet_*XY*	Az *XY*. fejezet példaprogramjait tároló alkönyvtár
\Gyak*XY*_CPP	Az **F2. függelék** *XY*. gyakorlati foglalkozásának mintapéldai (*/MintaXY_...*) és feladatmegoldásai (*/GyakXY_...*).
\Tanari kezikonyv	Az **F2. függelék** gyakorlati foglalkozásokhoz átdolgozott ismeretanyaga és forrásprogramjai.
\Gyakorlat_*XY*.pdf	Az *XY*. gyakorlati foglalkozás tananyaga PDF-formátumban.
\Gyak*XY*_CPP	Az *XY*. gyakorlati foglalkozás mintapéldai (*/MintaXY_...*) és feladatmegoldásai (*/GyakXY_...*).

\Teszt

Az önálló értékelést és tanulást segítő alkalmazás, illetve feladatlapok PDF-formátumban.



Teszt

Témakörök

1. A C++ program szerkezete
2. Alaptípusok, változók és konstansok
3. Operátorok és kifejezések
4. A C++ nyelv utasításai
5. Származtatott adattípusok
6. Függvények
7. Névterek és modulok használata
10. Fájlkezelés struktúrák felhasználásával
12. Osztályok

Beállítás: Szintbeállítás

☑ Van időkorlát?

30 perc

3 kérdés

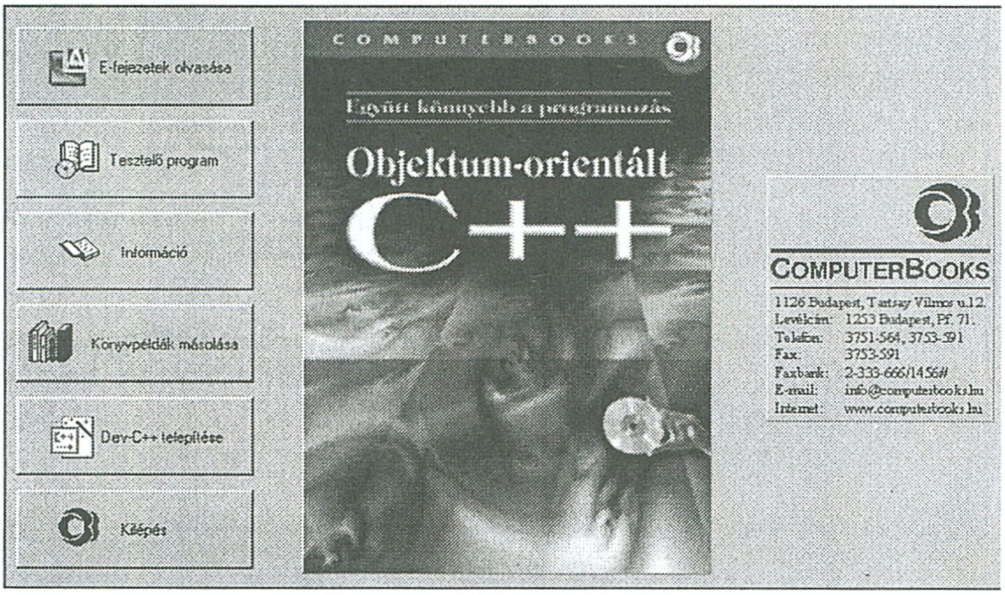
Mind
Egyik sem
Kezdő
Haladó
Szintbeállítás
Start
Kilép
Súgó

\Telepit

A könyvhöz tartozó forrásprogramok merevlemezre telepíthető formában

A CD használatát segítő segédprogramok és állományok

Fontos.exe



COMPUTERBOOKS

1126 Budapest, Tartsay Vilmos u.12.
Levélcím: 1253 Budapest, Pf. 71.
Telefon: 3751-564, 3753-591
Fax: 3753-591
Faxbank: 2-333-666/1456#
E-mail: info@computerbooks.hu
Internet: www.computerbooks.hu

E-fejezetek olvasása
Tesztelő program
Információ
Könyvpéldák másolása
Dev-C++ telepítése
Kilépés
COMPUTERBOOKS
Együtt könnyebb a programozás
Objektum-orientált
C++

CDTartalom.exe

A CD-melléklet tartalomjegyzéke
Kezdőlap
Tartalomjegyzék
Fejezetek
Kilépés
COMPUTERBOOKS
Együtt könnyebb a programozás
Objektum-orientált
C++

Fontos.htm

Tárgymutató

A

absztrakt osztály, 323
absztrakt tagfüggvény, 323
adatbevitel, 133
adatkivitel, 133
adatkódolás, 16
adatrejtés, 113
adattag, 113, 117
    dinamikus, 131
alapértelmezés szerinti argumentum, 267
alaposztály, 113, 307
algoritmusok, 91
állomány nyitása írásra, 275
állomány nyitása olvasásra, 275
állomány végének vizsgálata, 275
általános típus, 60
általánositott függvények, 80
általánosított osztály, 331
append, 279
aritmetikai operátorok, 228
átdefiniált adattípus, 51
automatikus konverzió, 246

B

barát függvény, 122
barát mechanizmus, 287
barátosztály, 288
base class, 307
beágyazás, 146, 330
bemenő paraméter, 257
beszúrás a láncba, 162
betűérzékeny, 227
bezáródás jelensége, 140
bináris állomány, 107
bináris fa, 204
bináris fájlkezelés, 281
biteltoló művelet, 16, 236
bitműveletek, 15, 235
blokk, 29
break, 35, 40, 204, 238, 241

C

case, 35
cast, 11
cath, 348
cerr, 228, 253
char, 9
char tömb, 50
ciklus utasítás, 38, 240
cin, 5, 27, 228
cin.get(), 10
class generator, 331
close, 108
cmath, 22
const, 10
continue, 39, 241
copy constructor, 295
cout, 5, 7, 228
cout.setf, 24
csomópontok, 204
cstdarg, 267
cstdlib, 179

D

data hiding, 113
data member, 117
decrement, 234
default argumentum, 74
default paraméter, 267
definíció, 7, 257
deklaráció, 7
delete, 21, 52, 55, 128, 248, 300
derived class, 307
destruktor, 113, 125, 128, 294
dinamikus objektumpéldány, 130
dinamikus struktúra, 253
do - while utasítás, 241
double, 7, 10
double*, 10, 12
double**, 12
do-while, 43

E

early binding, 113, 141, 321
egész konstansok, 227
egy karakter írása, 275
egy karakter olvasása, 275
egydimenziós tömb, 45
egységbezárás, 284
encapsulation, 113, 284
endl, 5, 7
enum, 8
erase(), 203
értékadás, 13
extern, 83, 85
extern tárolási osztály, 228

F

fail, 107
fájlkezelő függvények, 275
fastruktúra, 204, 207
feltételes operátor, 18, 237
float, 7
for, 40, 42, 50
for ciklus, 241
főátlóbeli elem, 49
friend, 175, 176, 183, 189, 287, 311
friend mechanizmus, 122
függvény, 59, 257
függvény átdefiniálása, 269
függvény deklarációja, 59
függvény paraméterlista, 257
függvényhívás, 61
függvényre mutató pointer, 262

G

generic class, 331
getline(), 50, 277
goto, 241

H

halmaztípus, 173, 177
hatókör operátor, 19
hexadecimális, 26
hibastream, 228
hozzáadás a lánc elejére, 342
hozzáadás a lánc végére, 342
hozzáférés
  private, 117, 179
  protected, 117, 137
  public, 117
hozzáfűzési mód, 279

I

if utasítás, 29
ifstream, 107, 279
implicit konverzió, 246
increment, 234
indexelő operátor, 193
információ elrejtése, 286
inheritance, 113, 137, 284, 307
inline, 125, 174, 190, 192, 257
inline függvény, 78
insert(), 203
instance, 117
int, 7
interpoláció
  Lagrange, 165
  lineáris, 165
ios
    fixed, 24
    scientific, 24
    showpos, 25
    left, 27
    right, 27
istream, 133, 301
iterátor függvény, 195

K

karakterkonstansok, 227
karaktersorozat, 50
karaktersorozat argumentumok, 271
késői kötés, 113, 141, 143, 322
kétirányú láncolt lista, 341
kifejezés, 29
kimenő paraméter, 257
kivételek, 227
kivételkezelés, 348
kódmakró, 257
konkatenálás, 191
konstans, 22
konstruktor, 113, 125, 293

  copy contructor, 126
  default, 126
  másoló, 126
  paraméteres, 126
korai kötés, 113, 140, 321
közös névterület, 333

L

láncolt lista, 103, 153, 338
late binding, 113, 141, 143, 322
lebegőpontos konstansok, 227
legnagyobb közös osztó, 94
léptető operátorok, 14, 234
lineáris interpoláció, 99
listasablon, 211
logikai konstansok, 227
logikai operátorok, 234
long, 8, 27

M

main() függvény, 5, 59, 114, 150, 228
main() függvény paraméterei, 76, 270
main() függvény visszatérési értéke, 270
másoló konstruktor, 295
mátrix, 49
megjegyzések, 227
member function, 117
moduláris programozás, 6
multiple inheritance, 113, 307
mutató, 245
mutatókonstans, 227

N

namespace, 83, 333
névterek, 5, 83
new, 21, 52, 54, 128, 300
new operátor, 248, 348
nothrow, 348
numerikus adatok írása, 278
numerikus adatok olvasása, 278
numerikus módszer, 165
nyelv alapelemei, 227

O

objektumok sokalakúsága, 141
objektumok zártsága, 321
objektumpéldány, 117, 120
ofstream, 107, 133, 279
oktális, 26
operandus, 13
operátor, 13
operátor átdefiniálás, 113
operátor függvények, 133
operator overloading, 187, 284, 299
operátor túlterhelése, 299
operator+(), 182
operátorfüggvény, 175
ostream, 301
osztályok, 117
osztálysablon, 211
overloading, 79, 113

Ö

öröklés, 113, 137, 284, 307

P

paraméterek, 64
  alapértelmezés szerinti, 74
  aritmetikai típusú, 64
  felhasználói típusú, 65
  függvény, 73
  sztring, 72
  tömb, 66
  változó hosszúságú, 75
paraméterezett osztály, 331
példányosítás, 149
pointer paraméter, 258
polymorphism, 113, 141, 284, 322
postfix, 50
pow, 22, 231
precedencia, 228
private, 121, 286
project, 89
protected, 286
prototípus, 59
public, 286
pure virtual, 323

Q

qsort, 179

R

read, 108
redefine, 113
referencia, 246
referencia típus, 258
register változó, 228
rekurzív függvény, 77, 268
rekurzív mód, 206
relációjelek, 234
rendezés
    beszúrás módszerrel, 97
    buborék módszerrel, 97
return, 35, 59, 60, 61, 121, 257
rövidzár hatás, 34

S

sablon, 151, 227
sablonosztály, 149, 331
sablontár, 198
saját típus, 23
setw(), 26
short, 8
short int, 15
sizeof, 17, 46, 236
sokalakúság, 113, 284, 322
sqrt, 231
static változó, 228
statikus adattag, 325
statikus helyfoglalású objektumpéldány, 123
std, 5
stdlib.h, 33
strcat, 192
strcmp, 192
string típus, 201
strlen(), 190
struct, 51, 251
struktúra, 251
struktúra átadása függvénynek, 261
struktúratömb, 53
switch, 34, 77, 114, 238
system("PAUSE"), 33
szám összes osztói, 91
szám törzstényezői, 92
származtatás szintaxisa
    private, 137
    protected, 137
    public, 137
származtatott osztály, 113, 307
sztring, 50
sztringek összefűzése, 191
sztringkapcsolás, 187
sztringkonstansok, 227
sztring-összehasonlítás, 187

T

tagfüggvény, 113, 117
template, 80, 149, 331
this, 123, 125, 175, 180, 183, 191
this pointer, 286
throw, 348
típuskonverzió, 11, 246, 301
típusmódosítás, 20
tiszta virtuális függvény, 323
tokenek, 227
tolower, 195
toupper, 195
több modulból álló program, 87, 89
többdimenziós tömb, 49
többirányú elágazatatás, 238
többszörös öröklés, 307
tömb, 45
tömbök átadása függvénynek, 260
tömbváltozó, 247
törlés a lánc közepéből, 343
törlés a láncból, 159
try, 348
type_info, 247
typedef, 9, 251
typeid operátor, 247

U

UniCode karakterek, 227
unióképzés, 181
unsigned, 8
using, 84, 85
utasítás, 29

Ü

üres utasítás, 29

V

változó hosszúságú paraméterlista, 267
vector osztály, 201
vessző operátor, 18
virtual, 113, 143, 322
virtuális, 113
virtuális függvények, 322
void, 51, 60, 118, 123, 125
void típus, 258
void*, 11

W

while, 38
while utasítás, 240
with(n), 27
write, 107

Megrendelőlap
info@computerbooks.hu
Telefon. 3751-564, 3753-4591

... *Dr.Kovács T.– Dr.Kovácsné C. J.– Ozsváth M.:* **Adatkezelés az MS ACCESS 2000 alkalmazásával**	**3.497.-**
... *Kovács L.:* **Adtatbázisok tervezésének és kezelésének módszertana**	**4.200.-**
.... *Abonyi J. és társai .* **Adatbányászat a hatékonyság eszköze**	**4.900.-**
... *Czenky M.* **Adadtmodellezés * SQL és Access alkalmazás * SQL Server ADO ◎**	**4.552.-**
... *Dr. Hatvany Béla Csaba*: **ASP.NET vezérlők programozása**	**5.990.-**
... *Pintér M.:* **AutoCAD 2004 felhasználói ismeretek**	**4.919.-**
... *Móricz Attila:* **Tippek a CD íráshoz -◎**	**2.500.-**
... *Kuzmina J.-Dr.Tamás P.-Tóth B.:* **Windows alkalmazások fejlesztése C++Builder 6 rendszerben (e-könyv CD-n)**	**3900.-**
... *Benkő L.–Benkő T.né–Tóth B.:* **Programozzunk C nyelven – ◎**	**2.968.-**
... *Benkő T.né–Poppe A.*: **Együtt könnyebb a programozás - C – ◎**	**3.220.-**
... *Benkő T.né– Tóth B. .*: **Együtt könnyebb a programozás JAVA – ◎**	**3.990.-**
... *Tóth B.:* **Programozzunk C++ nyelven ! ◎**	**2.994.-**
... *Benkő T.né–Poppe A.*: **Együtt könnyebb a programozás - Objektum-orientált C++ -◎**	**3.217.-**
... *Kőhalmi Éva- Kőhalmi Mariann:* **CorelDraw 12**	**5.590.-**
... *László József:* **Dinamikus weboldalak programozása - 💻**	**3.150.-**
... *Kovalcsik Géza:* **EXCEL 2000 –◎**	**3.400.-**
... *Kovalcsik Géza:* **Az EXCEL programozása –◎**	**4.800.-**
... *Kovalcsikné Pintér Orsolya:* **Az EXCEL függvényei A-tól Z-ig**	**3.779.-**
... *László József:* **Hangkártya programozása** Pascal és Assembly nyelven – 💾	**1.900.-**
... *Szirmay-Kalos.:* **Háromdimenziós grafika, animáció és játékfejlesztés -◎**	**5.496.-**
... *Ács Zsolt*: **Linux operációsrendszer(váltás)** - Linux 9 adminisztrátori ism. ◎	**2.979.-**
... *Czenky M.-Tamás P.- Vágási J.:* **Tanuljuk együtt az informatikát** ECDL elméleti modul – interaktív ◎ melléklettel	**3.495.-**
... *Dr.Kovácsné C.J.–Ozsváth M.–G.Nagy J.:* **Mit kell tudni Az EDCL vizsgához?** - Operációs rendszer, Szövegszerkesztés, Táblázatkezelés, Prezentáció, Internet és levelezés	**2.995.-**
... *Dr.Kovács T.-Dr.Kovácsné C.J.-Ozsváth M.-G.Nagy J.:* **Mit kell tudni? a PC-ről – Példatár**	**2.990.-**
... *László József:* **Mindenkinek az Internetről**	**1.999.-**
... *Móricz A.:* **Mobiltelefon a kezekben**	**990.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE 2000**	**2.093.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE XP**	**3.498.-**
... *Móricz A.* **Programozási alapfeladatok** - WAP oldalakhoz és WEB lapokhoz	**3.398.-**
... *Tóth B-. Benkő T.-né és* társai.: **Programozzunk Turbo Pascal nyelven – ◎**	**2.992.-**
... *Tamás P. – Tóth B. és* társai.: **Programozzunk DELPHI 7 rendszerben – ◎**	**3.860.-**
... *Benkő T.né.:* **Programozási feladatok és algoritmusok DELPHI rendszerben – ◎**	**2.999.-**
... *Juhász Tibor-Kiss Zsolt:* **Tanuljunk programozni** - Visual Basic Script, Objektumok, Web - interaktív ◎ melléklettel	**3.925.-**
...*Köhalmi Éva –Kőhalmi Tünde.* **Photoshop CS2 alapok és trükkök**	**6.450.-**

... *László József:* **PC hardver programozás** - valós és védett módban	4.720.-
... *Tátrai T.:* **MS Project 2000**	4.500.-
... *Mudri István dr:* **QuarkXpress 5**	5.500.-
... *Dr. Hatvany Béla Csaba:* **Bit képek feldolgozása VISUL BASIC rendszerben - ◉** - (feladok C++ és Delphi nyelven)	3.500.-
... *Tamás P. – Tóth B. és* társai.: **Programozzunk Visual Basic rendszerben –◉**	3.500.-
... *Benkő T.né.:* **Programozási feladatok és algoritmusok Visual Basic rendszerben– ◉**	2.999.-
... *Stolnicki Gyula:* **SQL programozóknak ◉**	4.970.-
....*Nagy S.:* **Útlevél a digitális világhoz** - Nyilvános kulcsú tanúsítványok a gyakorlatban Kriptográfia, digitális aláírások, Hitelesítés szolgáltatás, Webáruházak biztonsága, Elektronikus bank biztonsága, Tanúsítványok használata	2.680.-.
....*Nagy S.:* **Elektronikus leveleink védelme** Titkosítás, Digitális aláírás, Szabványos tanúsítványok	2.650.-
...*Mudri István:* **Videoszerkesztés Adobe Premiere Programokkal Premiere Pro, Pro 1.5 Premier Elements**	4.350.-
... *Gottdank Tibor*: **Szemantikus WEB Bevezetés a tudásalapú Internet világába**	4.500.-
... *Gottdank Tibor*: **WEBszolgáltatások - ◉ ◉**	4.995.-
... *Móricz Attila:* **WEBDESIGN a gyakorlatban**	2.399.-
... *Móricz A.:* **WINDOWS tippek az újratelepítéshez**	2.900.-
... *Gerő Judit:* **WORD 2000 I. kötet** – kezdőknek és haladóknak	2.600.-
... *Gerő Judit:* **WORD 2000 II. kötet** – haladóknak	3.499.-
... *Gerő J.–Reich G.:* **WORD for WINDOWS 6.0** – magyar & angol	990.-

Menedzsment informatikai szakkönyvek:

... *Szerk.:Hetyei J.:* **ERP rendszerek Magyarországon a 21. században**	6.900.-
... *Szerk.:Hetyei J.:* **Vezetői döntéstámogató és elektronikus kereskedelmi rendszerek Magyarországon**	4.900.-
... *Szerk.:Hetyei J.:* **Pénzintézetek és állami intézmények információs rendszerei Magyarországon**	5.500.-

Gazdasági informatika:

... *Lévayné Lakner Mária:* **EXCEL táblázatkezelő a gyakorlatban** – 💾	1.200.-
... *Balogh Gábor:* **Visual Basic és EXCEL programozás** – 💾	1.900.-
... *Csala Péter–Csetényi Athur–Tarlós Béla:* **Informatika alapjai**	3.999.-

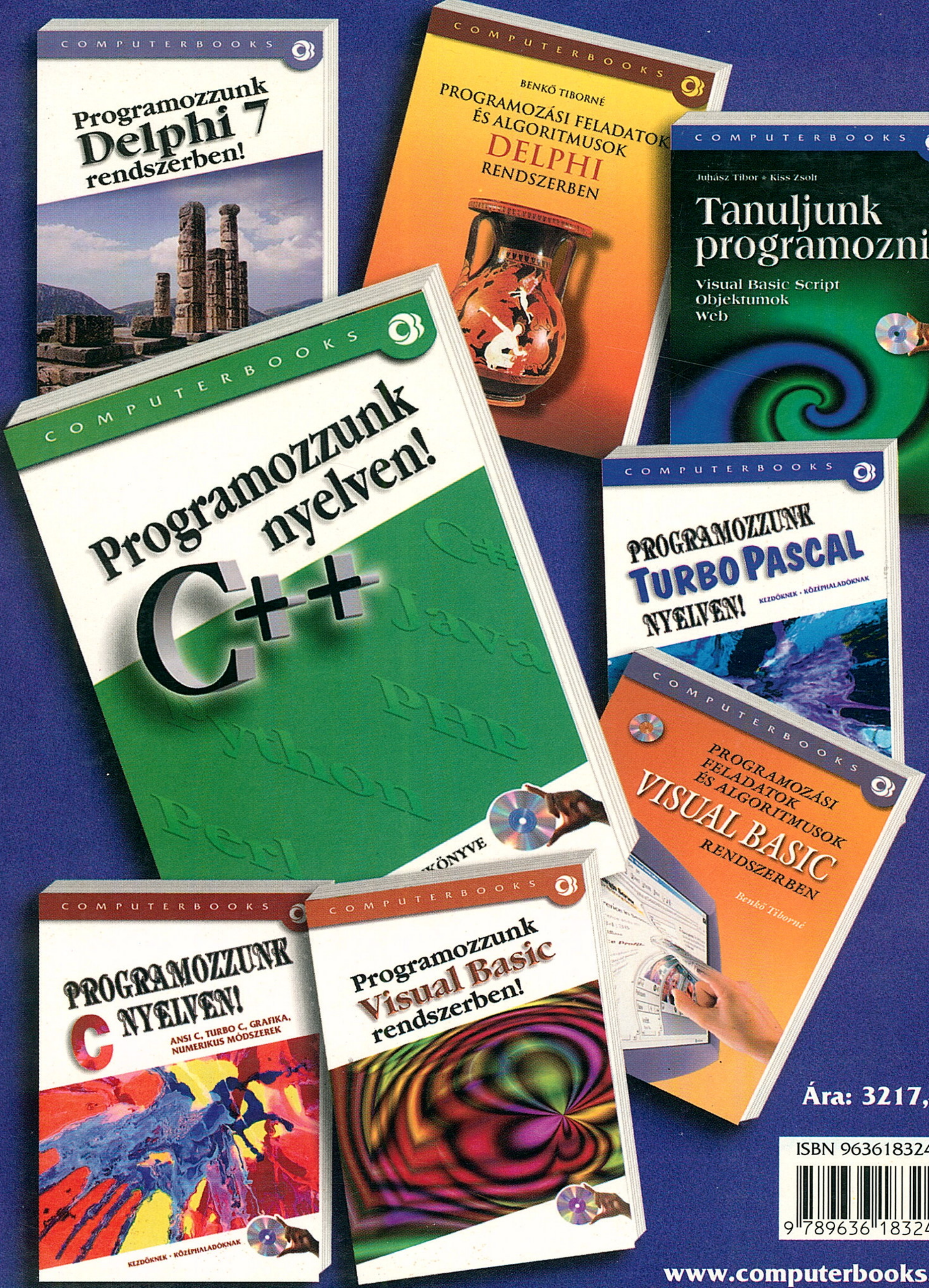
A Weboldalunkon részletes ismertetést talál könyveinkről:
www.computerbooks.hu

Megrendelő neve

Számlázási cím

Megjegyzés

PROGRAMOZZUNK... sorozatunk


COMPUTERBOOKS
Programozzunk Delphi 7 rendszerben!
BENKŐ TIBORNÉ
PROGRAMOZÁSI FELADATOK ÉS ALGORITMUSOK DELPHI RENDSZERBEN
Juhász Tibor • Kiss Zsolt
Tanuljunk programozni
Visual Basic Script
Objektumok
Web
Programozzunk C++ nyelven!
C# Java PHP Python Perl
KÖNYVE
PROGRAMOZZUNK TURBO PASCAL NYELVEN!
KEZDŐKNEK • KÖZÉPHALADÓKNAK
PROGRAMOZÁSI FELADATOK ÉS ALGORITMUSOK VISUAL BASIC RENDSZERBEN
Benkő Tiborné
PROGRAMOZZUNK C NYELVEN!
ANSI C, TURBO C, GRAFIKA, NUMERIKUS MÓDSZEREK
KEZDŐKNEK • KÖZÉPHALADÓKNAK
Programozzunk Visual Basic rendszerben!

Ára: 3217,–Ft

ISBN 963618324-4

9 789636 183240

www.computerbooks.hu